Vorlesungen über Betriebssysteme

Script zur Vorlesung

1992

Inhaltsverzeichnis

1 Grundbegriffe der Statistik **4**

1.1 Diskrete Zufallsvariablen 4

1.2 Stetige Zufallsvariablen 4

1.3 Erwartungswert einer Zufallsvariablen 5

1.4 Momente einer Zufallsvariablen 6

2 Die erzeugende Funktion **7**

3 Modellbildung und Systemanalyse **11**

3.1 Offene Systeme 11

3.1.1 Der Satz von Little 11

3.2 Das $M/G/1$-System 12

4 Konzepte und Analyse von Kommunikationsprotokollen **17**

4.1 Das ALOHA-System 17

4.1.1 Pure ALOHA 18

4.1.2 Slotted ALOHA 21

4.1.3 Slotted ALOHA mit endlich vielen Stationen . . 22

4.1.4 Die Kollisionsbehandlung 23

4.1.5 Gesamtübertragungszeit 24

4.2 CSMA-Protokolle 27

4.2.1 Non-persistent CSMA 27

4.2.2 1-persistent CSMA 28

4.2.3 p-persistent CSMA 28

4.2.4 Analyse 28

5 Das Problem des gegenseitigen Ausschlusses 32

5.1 Einführung 32

5.2 Synchronisierung nach Dekker ([Dij68]) 38

5.3 Synchronisation von N Prozessen 44

5.3.1 Modifikationen 50

5.3.2 Lösung nach Lamport 51

5.3.3 Lösung nach Peterson 52

6 Synchronisation mit Semaphoren [DIJK68] 54

6.1 Das Erzeuger-Verbraucher-Problem ([Dij68]) 56

6.2 Allgemeine/binäre Semaphore 59

6.3 Das Leser-Schreiber-Problem 61

6.3.1 Problem 1 61

6.3.2 Problem 2 62

6.4 Das 5-Philosophen-Problem 64

6.5 Implementierung von P- und V-Operationen 68

Abbildungsverzeichnis

3.1 Ankünfte und Abgänge im Warteschlangensystem 11

3.2 Während der Bedienzeit von c_n wartet bereits c_{n+1}. . . . 13

3.3 c_{n+1} kommt erst nach Ablauf der Bedienzeit von c_n. . . 13

4.1 Gefährdeter Bereich beim pure ALOHA. 19

4.2 Durchsatz beim pure ALOHA 20

4.3 Durchsatz beim slotted ALOHA 22

4.4 Durchsatz beim slotted ALOHA für verschieden viele Stationen. 23

4.5 Verzögerung beim slotted ALOHA. 26

4.6 Durchsatz beim Non-persistent CSMA 31

5.1 Präzedenzgraph 33

5.2 Modell eines Rechners mit einer und mit mehreren CPUs 34

1. Grundbegriffe der Statistik

In diesem Abschnitt soll ein kurzer Überblick über die Grundbegriffe der Statistik gegeben werden, soweit sie für die Analyse von Systemen im Bereich der Betriebssysteme benötigt werden. Damit wird gleichzeitig die verwendete Schreibweise eingeführt.

1.1 Diskrete Zufallsvariablen

Wir nennen eine Zufallsvariable X *diskret*, wenn X endlich viele Werte oder eine abzählbare Menge von Werten, $x_1, x_2, x_3, \ldots$, annehmen kann, so daß die Wahrscheinlichkeit des Auftretens eines bestimmten Wertes von X, nämlich x_i, als

$$p_i = \mathcal{P}[X = x_i]$$

mit $i = 0, 1, 2, \ldots$ definiert werden kann. Weiterhin muß

$$\sum_{i=0}^{\infty} p_i = 1$$

gelten.

Beispiele für diskrete Zufallsvariablen sind:

- Punktezahl beim Würfel.
- Anzahl der Prozesse in einem Betriebssytem.
- Anzahl der Druckaufträge vor einem Drucker.
- Anzahl der Kollisionen in einem lokalen Rechnernetz.

1.2 Stetige Zufallsvariablen

Wenn $\mathcal{P}[X = x_i] = 0$ für alle Werte von x_i gilt, dann ist X *stetig* (oder *kontinuierlich*). Hier hat es also keinen Sinn zu fragen, wie groß die Wahrscheinlichkeit für das Auftreten eines bestimmten Wertes von X, nämlich x_i, ist.

Wenn es eine nichtnegative Funktion $p(x)$ im Bereich $-\infty < x < \infty$ gibt, so daß es eine *Verteilungsfunktion*

$$P(x) = \int_{-\infty}^{x} p(x)\,dx$$

gibt, dann nennt man $p(x)$ die *Wahrscheinlichkeitsdichte* von X. Man kann die Verteilungsfunktion auch als

$$P(x) = \mathcal{P}[X \leq x]$$

formulieren, wie aus der Interpretation obigen Integrals folgt.

Die Verteilungsfunktion hat folgende Eigenschaften:

- $P(x)$ ist eine nichtfallende Funktion von x.
- $\lim_{x\to\infty} p(x) = p(\infty) = 1$.
- $\lim_{x\to-\infty} p(x) = p(-\infty) = 0$.

Weiter gilt

$$\mathcal{P}[a < X \leq b] = p(b) - p(a)$$

Beispiele für stetige Zufallsvariablen sind etwa:

- Rechenzeit eines Prozesses.
- Wartezeit auf die Beendigung eines Druckauftrages.
- Zeit zwischen zwei Tastatureingaben.
- Zeit zwischen zwei `login`-Prozessen.

1.3 Erwartungswert einer Zufallsvariablen

Wenn X eine diskrete Zufallsvariable ist, wobei jeder mögliche Wert $x \in X$ mit der Wahrscheinlichkeit $p(x) = \mathcal{P}[X = x]$ auftritt, dann ist der *Erwartungswert* von X

$$E[X] = \sum_{x:p(x)>0} x \cdot p(x).$$

Der Erwartungswert ist also das gewichtete Mittel über alle möglichen Werte, die X annehmen kann.

Ein empirischer Erwartungswert kann z.B. bei N Ereignissen, die jeweils einen Zufallswert $x \in X$ haben, durch

$$E[X] = \frac{\sum_{t=1}^{N} X_t}{N} = \sum_{i=1}^{N} \frac{i \cdot N(i)}{N} = \sum_{i=1}^{N} i \cdot \frac{N(i)}{N}$$

berechnet werden, wobei X_t die Zufallswerte sind, die die Werte $i \in N$ annehmen können, und wobei $N(i)$ die Anzahl des Auftretens von $X_t = i$ bedeuten. $N(i)/N$ kann man als die empirische Wahrscheinlichkeit für das Auftreten des Zufallswertes $i \in X$ deuten, mit

$$\lim_{N\to\infty} \frac{N(i)}{N} = p(i).$$

Wenn X ein Zufallswert aus $\mathbb{R}$ ist, dann ist der Erwartungswert

$$E[X] = \int_{-\infty}^{+\infty} x \cdot p(x)\,dx = \int_{-\infty}^{+\infty} x \cdot \frac{dP(x)}{dx}\,dx$$

1.4 Momente einer Zufallsvariablen

Das m-*te Moment* einer Zufallsvariablen ist wie folgt definiert:

$$E[X^m] = \sum_x x^m \cdot p(x), \text{ für } X \in I\!N \tag{1.1}$$

bzw.

$$E[X^m] = \int_{-\infty}^{+\infty} x^m \cdot p(x)\, dx, \text{ für } X \in I\!R \tag{1.2}$$

Das erste Moment, d.h. $m = 1$, von X ist der bereits genannte Erwartungswert von X, auch Mittelwert genannt. Das m-te zentrale Moment bezieht sich auf die Abweichung der einzelnen Zufallswerte vom Erwartungswert $E[X]$. Das erste zentrale Moment ist offensichtlich Null. Das zweite zentrale Moment ist die *Varianz* von X:

$$\begin{aligned} \sigma_X^2 &= E[(X - E[X])^2] \\ &= E[X^2 - 2X\,E[X] + E[X]^2] \\ &= E[X^2 - 2E[X]^2 + E[X]^2], \end{aligned}$$

also

$$\sigma_X^2 = E[X^2 - E[X]^2] = E[X^2] - E[X]^2. \tag{1.3}$$

Oft findet man auch die Schreibweise

$$\sigma_X^2 = \overline{x^2} - \overline{x}^2. \tag{1.4}$$

Sei X eine Zufallsvariable und g eine Funktion, dann ist auch $Y = g(X)$ eine Zufallsvariable. Wenn X eine diskrete Zufallsvariable mit den möglichen Werten $x_0, x_1, x_2, \ldots$ ist, dann ist der Erwartungswert von $g(X)$ gegeben durch

$$E[g(X)] = \sum_{i=0}^{\infty} g(x_i) \cdot \mathcal{P}[X = x_i], \tag{1.5}$$

und wenn X kontinuierlich ist, durch

$$E[g(X)] = \int g(x) \cdot p_X(x)\, dx. \tag{1.6}$$

2. Die erzeugende Funktion

Die *(moment-)erzeugende Funktion* findet Anwendeung bei der Transformation diskreter Zufallsvariablen. Sei X eine solche diskrete Zufallsvariable, die die Werte $x = 0, 1, 2, \ldots$ mit jeweils der Wahrscheinlichkeit $p_0, p_1, p_2, \ldots$ annehmen kann. Dann definiert man die erzeugende Funktion für die Variable X durch

$$G_X(z) = \sum_{k=0}^{\infty} p_k \cdot z^k \tag{2.1}$$

mit $p_k = \mathcal{P}[X = k]$. $G_X(z)$ ist offensichtlich eine Funktion nur von z, da über den Index k summiert wird und dieser Index in $G_X(z)$ explizit nicht mehr vorkommt.

Man kann die Transformation auch anders interpretieren. Vergleicht man Gleichung 2.1 mit den Gleichungen über den Erwartungswert der Zufallsvariable X, dann gilt offensichtlich (vergl. Gleichung 1.5)

$$G_X(z) = E[z^X] = \sum_{k=0}^{\infty} \mathcal{P}[X = k] \cdot z^k$$

Da laut Voraussetzung $\mathcal{P}[X = k] = p_k > 0$ und $\sum_{k=0}^{\infty} p_k = 1$ ist, ist $G_X(z)$ mindestens für $|z| \leq 1$ definiert und für $|z| < 1$ unendlich oft differenzierbar (z ist eine komplexe Variable).

Beispiel

Die geometrische Verteilung ist durch

$$p_k = (1 - \varrho) \cdot \varrho^k, \text{ für } k \geq 0 \text{ und } 0 \leq \varrho \leq 1$$

definiert. Die zugehörige erzeugende Funktion ist

$$G(z) = (1 - \varrho) \cdot \sum_{k=0}^{\infty} \varrho^k \cdot z^k = \frac{1 - \varrho}{1 - \varrho z}, \text{ mit } |\varrho z| < 1$$

Der Konvergenzradius ist also $|z| < 1/\varrho$.

Beispiel

Die Poissonverteilung ist durch

$$p_k = \frac{\lambda^k}{k!} e^{-\lambda}, \text{ für } k \geq 0$$

gegeben. Unter Verwendung der Gleichung

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

findet man

$$\begin{aligned} G(z) &= \sum_{k=0}^{\infty} \frac{\lambda^k}{k!} \cdot e^{-\lambda} \cdot z^k = e^{-\lambda} \cdot \sum_{k=0}^{\infty} \frac{(\lambda z)^k}{k!} \\ &= e^{\lambda(z-1)}, \text{ für } |z| < \infty \end{aligned}$$

Die Rücktransformation kann auch hier wie bereits bei der (noch allgemein gehaltenen) Z-Transformation durchgeführt werden. Wenn es jedoch nicht erforderlich ist, die einzelnen p_k zu kennen, wenn es genügt, die Momente der Zufallsvariablen X zu wissen, ist eine Rücktransformation nicht erforderlich.

Es gilt $G(z=0) = p_0$ und $G(z=1) = 1$.

Die erste Ableitung von $G(z)$ errechnet sich zu

$$\frac{dG_X(z)}{dz} = \sum_{k=0}^{\infty} \frac{d}{dz} \cdot p_k \cdot z^k = \sum_{k=0}^{\infty} k \cdot p_k \cdot z^{k-1}$$

mit $p_k = \mathcal{P}[X = k]$. An der Stelle $z = 1$ wird offensichtlich

$$\left.\frac{dG_X(z)}{dz}\right|_{z=1} = \sum_{k=0}^{\infty} k \cdot p_k = E[X]$$

Für die zweite Ableitung wird

$$\frac{d^2G_X(z)}{dz^2} = \sum_{k=0}^{\infty} k(k-1) \cdot p_k \cdot z^{k-2},$$

und bei $z = 1$ gilt

$$\frac{d^2G_X(z)}{dz^2} = E[X(X-1)] = E[X^2] - E[X].$$

Allgemein gilt

$$\begin{aligned} G_X^{(n)} &= \sum_{k=0}^{\infty} k(k-1)(k-2)\ldots(k-n+1)p_k \\ &= E[X(X-1)(X-2)\ldots(X-n+1)]. \end{aligned}$$

Aus dieser Gleichung, die die bereits angegebenen enthält, kann man die höheren Momente berechnen.

Beispiel

Die geometrische Verteilung kann durch

$$G_X(z) = \frac{1-\varrho}{1-\varrho \cdot z}$$

dargestellt werden. Für den Mittelwert findet man

$$\frac{dG_X(z)}{dz} = (1-\varrho) \cdot \varrho \cdot (1-\varrho \cdot z)^{-2}$$

und bei $z = 1$ gilt

$$E[X] = (1-\varrho) \cdot \varrho \cdot (1-\varrho)^{-2} = \frac{\varrho}{1-\varrho}$$

Das zweite Moment findet man aus

$$\frac{d^2G_X(z)}{dz^2} = (1-\varrho) \cdot \varrho \cdot (1-\varrho \cdot z)^{-3} \cdot 2\varrho$$

bei $z = 1$ gilt

$$E[X(X-1)] = \frac{2\varrho^2}{(1-\varrho)^2}$$

Daraus wird

$$E[X^2] - E[X] = E[X^2] - \frac{\varrho}{1-\varrho} = \frac{2\varrho^2}{(1-\varrho)^2}$$

also

$$E[X^2] = \frac{2\varrho^2}{(1-\varrho)^2} + \frac{\varrho}{(1-\varrho)} = \frac{\varrho \cdot (1+\varrho)}{(1-\varrho)^2}$$

Im nächsten Schritt wird die Varianz berechnet:

$$\begin{aligned} \sigma_X^2 &= E[X^2] - E[X] \\ &= \frac{\varrho \cdot (1+\varrho)}{(1-\varrho)^2} - \frac{\varrho^2}{(1-\varrho)^2} = \frac{\varrho}{(1-\varrho)^2} \end{aligned}$$

Beispiel

Die erzeugende Funktion der Poissonverteilung lautet

$$G_X(z) = e^{\lambda(z-1)}$$

Den Erwartungswert findet man über

$$\frac{dG_X(z)}{dz} = \lambda \cdot e^{\lambda(z-1)}$$

Bei $z = 1$ gilt

$$E[X] = \lambda$$

Das zweite Moment findet man über

$$\frac{d^2 G_X(z)}{dz^2} = \lambda^2 \cdot e^{\lambda(z-1)}$$

Es ist

$$E[X(X-1)] = E[X^2] - E[X] = \lambda^2$$

bzw.

$$E[X^2] = \lambda^2 + \lambda$$

Damit errechnet sich schließlich die Varianz zu

$$\sigma_X^2 = E[X^2] - E[X]^2 = (\lambda^2 + \lambda) - \lambda = \lambda$$

3. Modellbildung und Systemanalyse

3.1 Offene Systeme

3.1.1 Der Satz von Little

Eine der einfachsten und wichtigsten Gleichungen in der Theorie der Warteschlangen ist

$$L = \lambda \cdot T,$$

die als *Satz von Little* bekannt ist. Die Gleichung stammt zwar nicht von Little, er lieferte aber 1961 einen Beweis ([Lit61]).

Betrachten wir ein Warteschlangensystem, das Kunden betreten und nach einiger Zeit wieder verlassen. Wir beobachten Eingang und Ausgang und notieren jeweils, wann ein Kunde das System betritt und wann ein Kunde das System verläßt. Es gibt also zwei Zählprozesse, $A(t)$ (=Anzahl der bis zum Zeitpunkt t eingetroffenen Kunden) und $D(t)$ (=Anzahl der Kunden, die bis zum Zeitpunkt t das System verlassen haben). Die Anzahl der zur Zeit t im System befindlichen Kunden ist offensichtlich

$$L(t) = A(t) - D(t). \tag{3.1}$$

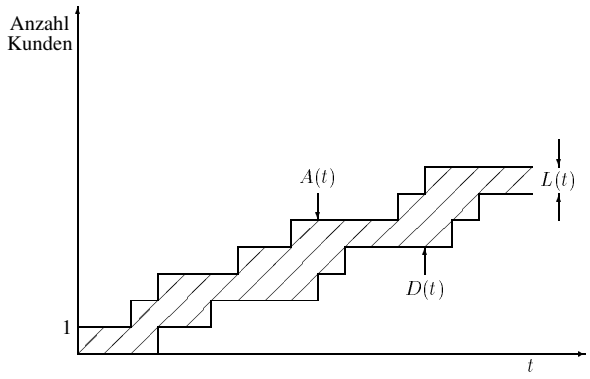

Abbildung 3.1: Ankünfte und Abgänge im Warteschlangensystem

Unsere Buchführung erlaubt es uns, die mittlere Ankunftsrate für das Intervall $(0,t)$ zu berechnen:

$$\lambda_t = \frac{A(t)}{t} \tag{3.2}$$

Die Abgangsrate ergibt sich ebenfalls:

$$\varphi_t = \frac{D(t)}{t} \tag{3.3}$$

Wenn der Beobachtungszeitraum nun groß genug wird, d.h. $t \to \infty$, und die Anzahl der Kunden im System, $L(t)$, endlich bleibt, dann wird $\lambda_\infty = \varphi_\infty$, d.h. Zugangs- und Abgangsrate werden gleich.

Die im Bild 3.1 schraffierte Fläche kann man als bisher aufgelaufene Verweilzeit $\gamma(t)$ sämtlicher Kunden betrachten. (Wenn jetzt $L(t)$ Kunden im System sind, dann werden sie im Intervall $(t, t+\Delta t)$ weitere $L \cdot \Delta t$ Zeiteinheiten an Verweilzeit aufnehmen, sofern sich $L(t)$ in dieser Zeit nicht ändert.)

Die mittlere Verweilzeit eines Kunden erhält man, indem man die gesamte Verweilzeit, $\gamma(t)$, durch die Anzahl der Ankünfte dividiert.

$$T_t = \frac{\gamma(t)}{A(t)} \tag{3.4}$$

Andererseits erhält man die mittlere Anzahl Kunden im System aus

$$\overline{L_t} = \frac{\gamma(t)}{t} \tag{3.5}$$

Faßt man Gleichung 3.2, Gleichung 3.4 und Gleichung 3.5 zusammen, so wird

$$\overline{L_t} = \lambda_t \cdot T_t,$$

und nach dem Grenzübergang $t \to \infty$ erhält man

$$L = \lambda \cdot T,$$

mit $\lambda = \lim_{t\to\infty} \lambda_t$ und $T = \lim_{t\to\infty} T_t$.

3.2 Das $M/G/1$-System

Da die Annahme einer Exponentialverteilung für den Bedienprozeß nicht sehr realistisch ist (bei bestimmten Scheduling-Algotithmen kann von einer zufällig verteilten Ankunft in der BE keine Rede sein), soll hier ein $M/G/1$-System untersucht werden. Die Verteilung des Bedienprozesses ist also beliebig.

Auch hier soll berechnet werden, wieviele Kunden sich im Mittel im System befinden. Dazu betrachtet man das System nur zu den Zeitpunkten, an denen ein Kunde die BE verläßt. Es gibt dabei zwei Fälle zu unterscheiden, die in Bild 3.2 und Bild 3.3 dargestellt sind.

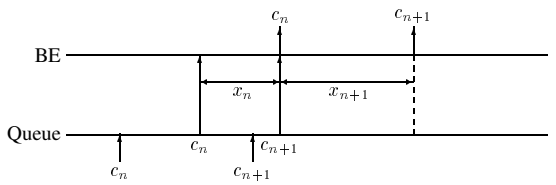

Abbildung 3.2: Während der Bedienzeit von c_n wartet bereits c_{n+1}.

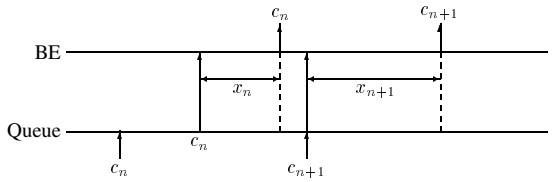

Abbildung 3.3: c_{n+1} kommt erst nach Ablauf der Bedienzeit von c_n.

1. Während der Bedienzeit des n-ten Kunden c_n wartet bereits mindestens ein weiterer Kunde c_{n+1}. Es gilt dann:

$$q_{n+1} = q_n - 1 + v_{n+1} \qquad (q_n > 0)$$

2. Während der Bedienzeit des n-ten Kunden c_n ist kein weiterer Kunde da, erst danach kommt c_{n+1} ins System. Dann gilt:

$$q_{n+1} = v_{n+1} \qquad (q_n = 0)$$

Dabei sei:

- c_n: n-ter Kunde.
- q_n: Anzahl der Kunden, die c_n hinter sich im System läßt.
- x_n: Bedienzeit des Kunden c_n.
- v_n: Anzahl der Ankünfte während der Bedienzeit von c_n.

Aus den beiden Fällen ergibt sich für q_{n+1}:

$$q_{n+1} = \begin{cases} q_n - 1 + v_{n+1}\,, & \text{für } q_n > 0 \\ v_{n+1}\,, & \text{für } q_n = 0 \end{cases}$$

Um diese Gleichungen zusammenzufassen, benutzt man die Sprungfunktion Δ_K[1]

$$\Delta_K = \begin{cases} 1, \text{ für } K = 1, 2, 3, \ldots \\ 0, \text{ für } K = 0 \end{cases}$$

und erhält damit

$$q_{n+1} = q_n - \Delta_{q_n} + v_{n+1}.$$

Der erste Versuch besteht im Bilden des Erwartungswertes:

$$E[q_{n+1}] = E[q_n] - E[\Delta_{q_n}] + E[v_{n+1}]$$

Da $n \to \infty$ und $\lim_{n\to\infty} E[q_{n+1}] = E[q_n]$ (stationärer Fall), folgt:

$$\begin{aligned} E[q] &= E[q] - E[\Delta_{\overline{q}}] + E[v] \\ \iff 0 &= -E[\Delta_{\overline{q}}] + E[v] \\ \iff E[\Delta_{\overline{q}}] &= E[v] \end{aligned}$$

(bei der Sprungfunktion muß man hier $\overline{q}$ als Index nehmen, da es q_∞ nicht gibt.)

Der gesuchte Erwartungswert ist zwar weggefallen, aber man kann immerhin die mittlere Anzahl von Ankünften pro Bedienzeit ausrechnen:

$$\begin{aligned} E[v] = E[\Delta_{\overline{q}}] &= \sum_{K=0}^{\infty} \Delta_K \cdot P[q = K] \\ &= \Delta_0 \cdot P[q = 0] + \Delta_1 \cdot P[q = 1] + \Delta_2 \cdot P[q = 2] + \ldots \\ &= 0 + 1 \cdot (P[q = 1] + P[q = 2] + \ldots) \\ &= P[q > 0] \\ &= 1 - P[q = 0] \\ &= \varrho = \lambda \cdot \overline{x} \end{aligned}$$

Das Ergebnis hätte auch einfacher gewonnen werden können. $E[v]$ sagt nämlich, wieviele Kunden (im Mittel) während einer (mittleren) Bedienzeit eintreffen. Für den Ankunftsprozeß wurde bereits $E[v] = \lambda \cdot t$ hergeleitet, wobei t eine beliebige Zeit war, hier jedoch die mittlere Bedienzeit sein soll, woraus $E[v] = \lambda \cdot \overline{x}$ folgt. Diese vereinfachte Mitelwertbildung ist erlaubt, da λ eine Konstante ist und die Ankünfte unabhängig vom Geschehen in der BE erfolgen.

Da der erste Ansatz nicht zur Lösung führt, quadriert man die Gleichung vor dem Bilden des Erwartungswertes:

$$\begin{aligned} (q_{n+1})^2 &= (q_n - \Delta_{q_n} + v_{n+1})^2 \\ &= (q_n)^2 + (\Delta_{q_n})^2 + (v_{n+1})^2 - 2 \cdot \Delta_{q_n} \cdot v_{n+1} \end{aligned}$$

[1] Wie man aus der Definition von Δ_K sieht, muß der Erwartungswert $E[\Delta_K] < 1$ sein.

$$\begin{aligned} &-2 \cdot q_n \cdot \Delta_{q_n} + 2 \cdot q_n \cdot v_{n+1} \\ = \; &(q_n)^2 + \Delta_{q_n} + (v_{n+1})^2 - 2 \cdot \Delta_{q_n} \cdot v_{n+1} \\ &-2 \cdot q_n + 2 \cdot q_n \cdot v_{n+1} \qquad (\text{da } (\Delta_{q_n})^2 = \Delta_{q_n} \text{ und } q_n \cdot \Delta_{q_n} = q_n) \end{aligned}$$

Bilden des Erwartungswertes und $n \to \infty$ ergibt nun:

$$\begin{aligned} E[q^2] &= E[q^2] + E[v^2] + E[\Delta_{\overline{q}}] - E[2 \cdot \Delta_{\overline{q}} \cdot v] \\ &\quad -E[2 \cdot q] + E[2 \cdot q \cdot v] \\ \iff 0 &= E[v^2] + E[\Delta_{\overline{q}}] - 2 \cdot E[\Delta_{\overline{q}}] \cdot E[v] \\ &\quad -2 \cdot E[q] + 2 \cdot E[q] \cdot E[v] \\ &= E[v^2] + \varrho - 2 \cdot \varrho \cdot \varrho - 2 \cdot E[q] + 2 \cdot E[q] \cdot \varrho \\ &= E[v^2] + \varrho - 2 \cdot \varrho^2 - E[q] \cdot (2 - 2 \cdot \varrho) \\ \iff E[q] \cdot (2 - 2 \cdot \varrho) &= E[v^2] + \varrho - 2 \cdot \varrho^2 \\ &= E[v^2] - \varrho + 2 \cdot \varrho - 2 \cdot \varrho^2 \\ &= E[v^2] - \varrho + \varrho \cdot (2 - 2 \cdot \varrho) \\ \iff E[q] &= \frac{E[v^2] - \varrho}{2 \cdot (1 - \varrho)} + \varrho \\ &= \varrho + \frac{E[v^2] - E[v]}{2 \cdot (1 - \varrho)} \end{aligned}$$

In diesem Ausdruck ist nur $E[v^2]$ unbekannt, denn $E[v] = \varrho = \lambda \cdot \overline{x}$. Wie schon bei der Anmerkung zu $E[v]$ im ersten Ansatz kann man auch hier sagen, daß $E[v^2]$ für beliebige Zeiten, d.h. für den reinen Ankunftsprozeß, alleine aus der zugehörigen Poissonverteilung hergeleitet werden kann. Unter Verwendung der Definition für die Varianz, $\text{var}_X = E[X^2] - E[X]^2$ (Gleichung 1.3), findet man

$$E[v^2] = \sigma_v + E[v]^2$$

und speziell für die Poissonverteilung

$$E[v^2] = \lambda t + (\lambda t)^2$$

mit noch beliebigem t. Da aber in $E[v^2]$ die (mittlere) Zahl der Ankünfte—genauer: eine Funktion von v, nämlich v^2—während einer (mittleren) Bedienzeit $\overline{x}$ verlangt wird, setzt man die entsprechenden Werte ein und findet

$$E[v^2] = \lambda \cdot \overline{x} + \lambda^2 \cdot \overline{x^2},$$

da λ bei der Mittelwertbildung konstant ist und Ankunfts- und Bedienprozeß voneinander unabhängig sind. Eine ausführliche Berechnung erfolgt über

$$\begin{aligned} E[v^2]_{t=x} &= \int_{t=0}^{\infty} \text{var}_v \, dp + \int_{t=0}^{\infty} E[v]^2 \, dp \\ &= \int_{t=0}^{\infty} \lambda t \, dp + \int_{t=0}^{\infty} (\lambda t)^2 \, dp \quad (t = x). \end{aligned}$$

Setzt man also $E[v^2] - E[v] = \lambda^2 \cdot \overline{x^2}$ in den oben erhaltenen Ausdruck ein, ergibt sich die Pollaczek-Khinchin-Gleichung (PK-Gleichung):

$$E[q] = \varrho + \frac{\lambda^2 \cdot \overline{x^2}}{2 \cdot (1 - \varrho)} = \lambda \cdot \overline{x} + \frac{\lambda^2 \cdot \overline{x^2}}{2 \cdot (1 - \varrho)}$$

In der PK-Gleichung, die die mittlere Anzahl von Kunden im System angibt, werden von der Bedienzeitverteilung nur die ersten beiden Momente benötigt, $\overline{x}$ und $\overline{x^2}$. Setzt man noch $C^2 = (\overline{x^2} - \overline{x}^2)/(\overline{x}^2)$, und setzt dies ein, so erhält man:

$$E[q] = \varrho + \varrho^2 \cdot \frac{1 + C^2}{2 \cdot (1 - \varrho)}$$

$E[q]$ steigt also linear mit C^2 und ist am kleinsten für $C^2 = 0$, d.h. wenn die Varianz gleich 0 ist.

4. Konzepte und Analyse von Kommunikationsprotokollen

4.1 Das ALOHA-System

Anfang der siebziger Jahre wurde an der Universität von Hawai eine Technik entwickelt, mit deren Hilfe räumlich z.T. weit entfernte und voneinander unabhängig arbeitende Computer miteinander kommunizieren konnten. Das Verfahren wird *ALOHA-System* genannt. „ALOHA“ stammt aus dem Hawaianischen und ist laut [Bla89, S. 810] eine Grußformel „without regardings to whether a person is arriving or departing“, könnte also einfach mit „Hallo“ übersetzt werden. Die Datenübertragung erfolgte damals über Radiosender, wobei das Problem darin lag, daß alle Sendestationen, auf derselben Frequenz arbeiten, ohne sich über die Zugriffsberechtigung verständigen zu können.

Heute erfolgt die Datenübertragung über Satelliten. Im Prinzip ist das ALOHA-System auch in lokalen Rechnernetzen anwendbar, hier gibt es jedoch effizientere Verfahren. Obwohl das ALOHA-System in seiner ursprünglichen Form unabhängig davon ist, ob die Übertragung über Satellit oder über terristische Radiostationen erfolgt, spielen bei der Verfeinerung des Konzeptes die Unterschiede der beiden Übertragungsarten doch eine Rolle. So ist z.B. die Laufzeit einer Nachricht über Satellit um einige Zehnerpotenzen größer als bei der Übertragung über Radiostationen oder über Kabel.

Die Arbeitsweise des ALOHA-Systems ist die folgende:

Es gibt eine Menge von Stationen, die über eine bestimmte Frequenz sowohl senden als auch empfangen können. Bei der Übertragung über Satellit senden alle Stationen auf derselben Frequenz und empfangen alle auf einer gemeinsamen anderen Frequenz. Der Satellit empfängt die Signale auf der einen Frequenz und strahlt sie auf der anderen wieder ab. Die Stationen sind räumlich voneinander getrennt, wobei die Entfernung keine Rolle spielt. Die Stationen können z.B. wenige Meter auseinander stehen und mit einem Kabel verbunden sein, oder sie können auf verschiedenen Kontinenten stehen und die Daten über einen Satelliten austauschen. Es wäre nun eigentlich erforderlich, das exklusiv benutzbare Betriebsmittel „Übertragungskanal“ an die sendewilligen Stationen zu verplanen. Zum Austausch der Synchronisationsnachrichten benötigt man aber auch wieder ein exklusiv benutzbares Betriebsmittel, womit man es wieder mit derselben Problematik zu tun hat. Deswegen verzichtet man beim ALOHA-Verfahren von vornherein auf Kollisionsfreiheit der Nachrichten und nimmt in Kauf, „irgendwie“ mit den Kollisionen fertig zu werden.

Wir werden uns im folgenden ausschließlich mit dem heute üblichen ALOHA-Verfahren befassen, bei dem die Datenübertragung über einen geostationären Satelliten erfogt.

4.1.1 Pure ALOHA

Alle Stationen sind bezüglich der Benutzung des Übertragungskanals gleichberechtigt. Jede Station, die senden will, sendet. Wenn mindestens zwei Stationen zur etwa gleichen Zeit senden, wird es zu Kollisionen kommen, die Daten überlagern sich und sind damit zerstört. Solche Kollisionen müssen erkannt werden, damit der Absender die Daten noch einmal senden kann. Wir haben es also zunächst mit drei Problemen zu tun:

1. Neue Nachricht senden (Beim *pure* ALOHA jederzeit möglich).
2. Kollisionen erkennen, um eine erneute Übertragung zu veranlassen.
3. Nachricht erneut senden.

Das Erkennen von Kollisionen ist am einfachsten, wenn die Daten über Satellit übertragen werden[1]. Sender und Empfänger müssen nämlich beide im Sichtbereich des Satelliten liegen. Beide hören also am Ende der Übertragung (vom Sender zum Satelliten und zurück zur Erde) dieselbe Nachricht, korrekte Übertragung vorausgesetzt. Der Sender kann also die empfangene Nachricht mit der vom ihm abgeschickten vergleichen und so feststellen, ob die Übertragung erfolgreich war oder nicht. Eine spezielle Quittung des Empfängers an den Sender ist nicht erforderlich.

Bei nicht erfolgreicher Übertragung, die nur durch die Überlagerung mit anderen Nachrichten zustande gekommen sein kann[2], muß die Nachricht noch einmal übertragen werden. Es gibt mit Sicherheit wieder eine Kollision, wenn alle Stationen, deren Nachrichten zerstört wurden, sofort einen neuen Versuch starten. Um das zu verhindern, ist eine besondere Kollisionsbehandlung erforderlich. Ohne diese näher zu kennen, nur unter der Annahme, daß es sie gibt, können wir bereits eine erste Analyse vornehmen:

Die zu übertragenden Daten werden in gleichgroße Pakete unterteilt. Die Zeit zum Senden eines Paketes betrage P Zeiteinheiten. Um nicht mit absoluten Zeiten arbeiten zu müssen, werden alle Zeiten auf diese P Zeiteinheiten bezogen. Ziel der Analyse ist zunächst, den Durchsatz zu berechnen. Dazu werden die folgenden Größen definiert:

S : Durchsatz = mittlere Anzahl der erfolgreich übertragenen Pakete pro Paketzeit P. Offensichtlich ist $0 \leq S \leq 1$. Dabei spielt keine Rolle, wie oft eine Übertragung wiederholt werden mußte, und wieviel Zeit seit dem ersten Start des Paketes vergangen ist, bis es endlich erfolgreich beim Empfänger angekommen ist.

[1] Nur damit werden wir uns hier befassen. Kollisionserkennung bei Übertragung per Kabel wird bei lokalen Rechnernetzen, speziell beim Ethernet, behandelt.

[2] Wir setzen immer korrekt arbeitende Übertragungswege voraus.

G : mittlere Anzahl abgeschickter Pakete (erste oder erneute Übertragung) pro Paketzeit P. Es gilt natürlich $S \leq G$. Wir nehmen an, daß jedes Paket irgendwann einmal erfolgreich übertragen wird.

Der Zusammenhang zwischen S und G läßt sich durch

$$S = G \cdot \mathcal{P}[\text{erfolgreiche Übertragung}] \tag{4.1}$$

ausdrücken und besagt, daß ein Paket erfolgreich übertragen wurde, wenn schließlich einer der evtl. zahlreichen Versuche erfolgreich war. Die Wahrscheinlichkeit, eine kollisionsfreie Übertragung durchzuführen, muß noch ermittelt werden. Die Größen in Gleichung 4.1 kann man auch so deuten: Der Quotient S/G ist die Anzahl der erfolgreichen Übertragungen, bezogen auf die Anzahl der gestarteten Übertragungen. Der Kehrwert G/S ist dann die Anzahl der Versuche, die eine Station im Mittel für eine erfolgreiche Übertragung machen muß.

Im nächsten Schritt muß die Wahrscheinlichkeit für eine erfolgreiche Übertragung bestimmt werden. Es ist klar, daß während der Sendezeit eines Paketes kein weiteres Paket abgeschickt werden darf, wenn es nicht zu einer Kollision kommen soll. Aber das allein genügt nicht, denn auch ein Paket, das knapp P Zeiteinheiten zuvor abgeschickt wurde, bewirkt eine Kollision. Ein Paket, das kollisionsfrei übertragen werden soll, muß also $2P$ Zeiteinheiten auf dem Übertragungsmedium für sich haben. Dieses Zeitintervall wird gefährdeter Bereich genannt (Siehe Bild 4.1).

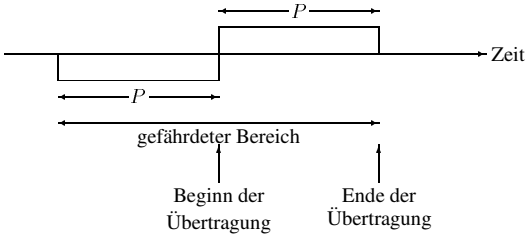

Abbildung 4.1: Gefährdeter Bereich beim pure ALOHA.

Wir nehmen nun an, daß die Sendevorgänge einen Poisson-Prozeß bilden, daß also die Zeiten zwischen zwei Sendevorgängen exponentiell verteilt sind. Aufgrund der Definition von G folgt, daß man G auch als Ankunftsrate des Poissonprozesses auffassen kann. Deswegen ist

$$\begin{aligned} & \mathcal{P}[\text{erfolgreiche Übertragung}] \\ = \; & \mathcal{P}[\text{im gef. Ber. wird kein weiteres Paket gesendet}] \\ = \; & e^{-(\text{Ankunftsrate})\cdot(\text{Zeitintervall})} \\ = \; & e^{-2G} \end{aligned}$$

(Hier ist die Ankunftsrate gleich G zu setzen und das Zeitintervall gleich 2, da 2 die Größe des gefährdeten Bereichs, normiert auf die Größe P, ist.)

Damit wird

$$S = G \cdot e^{-2G} \tag{4.2}$$

unter der Annahme, daß die Sendevorgänge einen Poissonprozeß bilden. Wir können eine Poisson-Prozeß annehmen, wenn folgende Bedingungen zutreffen:

- Die Stationen senden unabhngig voneinander.
- Es gibt (theoretisch) unendlich viele Stationen.
- Es geht kein Paket verloren. Jedes Paket wird irgenwann einmal erfolgreich übertragen.

Der maximale Durchsatz läßt sich aus Gleichung 4.2 leicht berechnen (nach G differenzieren und gleich 0 setzen). Man findet (siehe auch Bild 4.2):

$$S_{\max} = \frac{1}{2e} \approx 0.18 \text{ bei } G = \frac{1}{2} \tag{4.3}$$

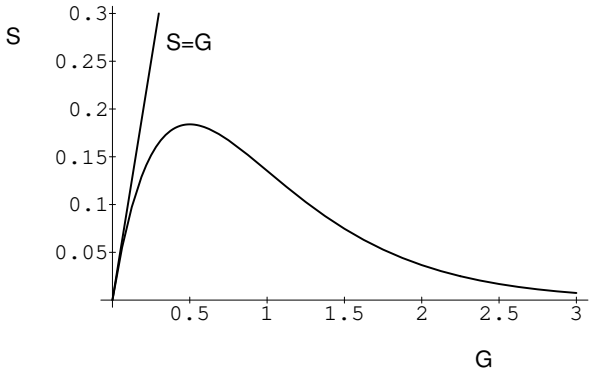

Abbildung 4.2: Durchsatz beim pure ALOHA

Man kann also den Übertragungskanal im günstigsten Fall zu nur 18% nutzen. Dieser günstigste Fall tritt ein, wenn $G = 1/2$ ist, wenn also im Mittel eine halbes Paket in der Zeit P gesendet wird, bzw. wenn im Mittel ein Paket in der Zeit $2P$, also im gefährdeten Bereich, gesendet wird. Daß bei $G = 1/2$ der Durchsatz optimal ist, hätte man auch durch etwas Nachdenken ohne Mathematik finden können: Es geht nämlich genau dann keine Kanalkapazität verloren, wenn unmittelbar nach $2P$ das

nächste Paket gesendet wird. Früher darf kein Paket kommen, da sonst Kollisionen entstehen.

Gleichung 4.3 sagt, daß beim maximalen Durchsatz der Quotient $S/G = 1/e$ ist, oder der Kehrwert $G/S = e \approx 2.7$, was bedeutet, daß im Mittel fast drei Versuche unternommen werden müssen bis ein Paket erfolgreich übertragen werden kann.

Eine wichtige Annahme bei der Herleitung von Gleichung 4.2 war die Existenz eines statistischen Gleichgewichtes. Diese Annahme ist aber nicht immer gültig, wie folgende Überlegung zeigt:

Nehmen wir einen Punkt auf der Kurve nach Gleichung 4.2 rechts vom Maximum an, es ist also $G > 1/2$. Lassen wir nun G aufgrund statistischer Schwankungen einen leicht veränderten Wert annehmen. Wenn G etwas größer wird, muß S kleiner werden. Der Durchsatz wird sich also verringern und die Zahl der Versuche G/S nimmt zu, wodurch G (als Ankunftsrate interpretiert) wiederum größer wird usw. Der Durchsatz geht also gegen Null und das Übertragungssystem bricht zusammen.

Wenn in dem angenommenen Punkt ($G > 1/2$) G etwas kleiner wird, dann wird S gröser und G/S kleiner, weshalb weniger Versuche notwendig sind. G wird wiederum kleiner und kann bis auf $G = 1/2$ absinken.

Diese Überlegungen zeigen, daß die in Bild 4.2 dargestellte Kurve rechts vom Maximum nicht stabil ist, sie kann in der Praxis nicht auftreten. Damit ist aber das letzte Wort über die Stabilität noch nicht gesprochen. Die Art der Kollisionsbehandlung beeinflußt nämlich das Stabilitätsverhalten.

4.1.2 Slotted ALOHA

Einen höheren Durchsatz kann man durch eine Modifikation des pure Aloha erzielen. Es ist jetzt den Sendern nicht mehr gestattet, zu jeder beliebigen Zeit zu senden, vielmehr dürfen sie nur getaktet senden, d.h. zu bestimmten Zeitpunkten, die äquidistant und gerade P Zeiteinheiten groß sind. Die Zeit wird also in sogenannte *slots* eingeteilt, die die Größe eines Paketes haben. Das Verfahren wird deshalb *slotted ALOHA* genannt.

Die Analyse verläuft genauso wie beim pure ALOHA. Der einzige Unterschied liegt in der Größe des gefährdeten Bereichs, der jetzt nur noch P Zeiteinheiten groß ist. (Kollisionen treten nur auf, wenn mind. zwei Stationen im gleichen Slot senden. Die Pakete dieses Slots sind dann alle zerstört.)

Die Analyse des slotted ALOHA führt zu

$$S = G \cdot e^{-G} \tag{4.4}$$

mit einem maximalen Durchsatz von (siehe Bild 4.3):

$$S_{\max} = \frac{1}{e} \approx 0.36 \text{ bei } G = 1 \tag{4.5}$$

Auch hier gilt, daß die Kurve rechts vom Maximum keine stabilen Zustände darstellt und in der Praxis nicht vorkommt.

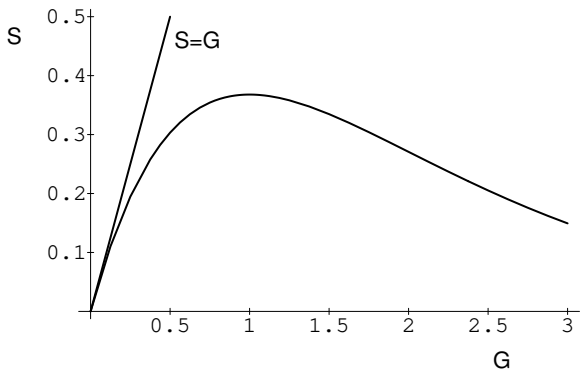

Abbildung 4.3: Durchsatz beim slotted ALOHA

4.1.3 Slotted ALOHA mit endlich vielen Stationen

Bei der bisherigen Analyse wurde eine unendliche Zahl von Stationen angenommen. Wir wollen jetzt die Analyse für eine endlich große Anzahl durchführen. Wir nehmen M voneinander unabhängige Sender in einem slotted ALOHA-System an. Jeder Sender m hat an der Gesamtzahl der abgeschickten Pakete einen Anteil G_m ($1 \leq m \leq M$):

$$G_m = \mathcal{P}\,[\text{Station } m \text{ sendet ein Paket}]$$

Der gesamte Verkehr im Übertragungskanal ist somit $G = \sum_{m=1}^{M} G_m$ Pakete pro Slot.

Für erfolgreich übertragene Pakete gilt analog

$$S_m = \mathcal{P}\,[\text{Paket der Station } m \text{ wurde erfolgreich übertragen}]$$

und $S = \sum_{m=1}^{M} S_m$.

Ein Paket kann nur dann erfolgreich übertragen werden, wenn nur eine Station sendet (mit der Wahrscheinlichkeit G_m) und alle anderen Stationen nicht senden (jeweils mit der Wahrscheinlichkeit $1 - G_i$, $i \neq m$). Es wird also

$$S_m = G_m \prod_{i \neq m} (1 - G_i), \text{ mit } 1 \leq m \leq M \tag{4.6}$$

Wenn sich alle Stationen statistisch gleich verhalten, dann ist $S_m = S/M$ und $G_m = G/M$. Das führt zu

$$S = G \cdot \left(1 - \frac{G}{M}\right)^{M-1} \tag{4.7}$$

(Für $M \to \infty$ geht Gleichung 4.7 übrigens wegen $\lim_{n\to\infty} \left(1 - \frac{x}{n}\right)^n = e^x$ in $S = G \cdot e^{-G}$ über.)

Aus Gleichung 4.7 sieht man auch, daß nur bei $M = 1$ der Idealfall $S = G$ auftreten kann und muß, denn bei nur einer einzigen Station kann keine Nachricht zerstört werden: Der maximale Durchsatz liegt nach Gleichung 4.7 bei $G = 1$ unabhängig von M und ist

$$S_{\max} = \left(1 - \frac{1}{M}\right)^{M-1}$$

In Bild 4.4 ist der Durchsatz für verschiedene Werte von M dargestellt, um zu sehen, wie weit unsere erste Annahme ($M \to \infty$) mit nur endlich vielen Stationen übereinstimmt. Das Bild zeigt, daß es legitim ist, mit dem mathematisch einfachsten Modell $M \to \infty$ zu rechnen. Die Ergebnisse „liegen auf der sicheren Seite“, denn sie sind „pessimistisch“. Die tatsächlichen Verhältnisse mit endlichem M führen zu einem besseren Durchsatz.

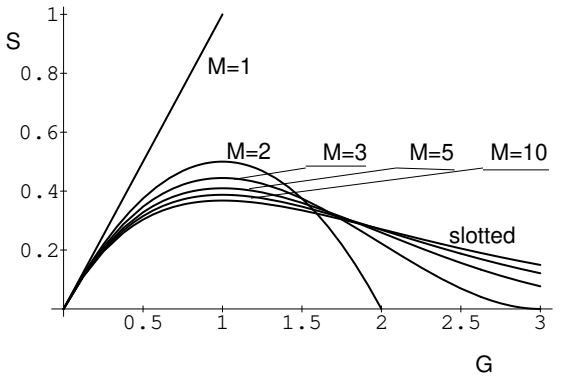

Abbildung 4.4: Durchsatz beim slotted ALOHA für verschieden viele Stationen.

4.1.4 Die Kollisionsbehandlung

Eine einfache Überlegung hat bereits gezeigt, daß es tödlich für das Übertragungssystem ist, wenn jeder Sender, der eine Zerstörung des von ihm gesendeten Paketes feststellt, sofort eine neue Übertragung startet. Da sich aber die an

der Kollision beteiligten Stationen nicht über einen neuen Sendezeitpunkt verständigen können[3], muß der Zufall die Rettung bringen:

Ein Sender, der ein Paket nocheinmal senden muß, erzeugt eine Zufallszahl k mit $1 \leq k \leq K$ und sendet das Paket nachdem er k Slots gewartet hat. Das Paket wird also innerhalb der nächsten K Slots gesendet, wobei alle Slots mit der gleichen Wahrscheinlichkeit $1/K$ ausgewählt werden. K ist eine fest wählbare Konstante, die jedoch hinreichend groß zu wählen ist, damit das Verfahren funktioniert. Der Einfluß von K wird im folgenden untersucht.

4.1.5 Gesamtübertragungszeit

Wir haben in der bisherigen Untersuchung gesehen, daß zwar eine kollisionsfreie Datenübertragung nicht gewährleistet ist, daß aber durch eine nachgeschaltete Kollisionsbehandlung mit erneutem Senden des Paketes mit einer von Null verschiedenen Wahrscheinlichkeit die Übertragung schließlich doch noch erfolgreich sein wird. Wann das sein wird wurde bisher nicht untersucht, soll jedoch in diesem Abschnitt nachgeholt werden. Die Frage lautet also: „Nach welcher Zeit kommt ein Paket (im Mittel) erfolgreich beim Empfänger an?“

Im Folgenden soll eine vereinfachte Analyse des Zeitverhaltens beim slotted ALOHA-System durchgeführt werden. Dazu sehen wir uns an, welchen Weg ein Paket zurückzulegen hat, bis es den Empfänger erfolgreich erreicht. Um auch mit Zahlen arbeiten zu können, hier die Kenndaten des ALOHA-Systems (Siehe [Kle76, S. 370]):

Die über den Satelltiten zu übertragenden Daten werden in Pakete der Größe 1125 Bits eingeteilt und mit 50 kBit/s übertragen. Die Paketzeit P beträgt also 22.5 ms. Dies ist die Mindestzeit für die Übertragung eines Paketes, wenn der Empfänger direkt neben dem Absender steht. Beim Weg über den Satelliten, der ca. 36.000 km über dem Äquator steht, hat das Paket einen Weg von mindestens 72.000 km zurückzulegen, wenn Sender und Empfänger am Äquator stehen. Der Weg verlängert sich um bis zu fast 12.000 km, wenn Sender und Empfänger den Satelliten am Horizont sehen, also nicht weit vom Nord- oder Südpol entfernt sind. Dieser Weg entspricht einer Laufzeit von 240 bis 280 ms. Nehmen wir eine mittlere Laufzeit von 270 ms an, dann können in dieser Zeit $R = 12$ Pakete abgeschickt werden. Nach der Zeit $P + R \cdot P$ ist das Paket wieder zurückgekommen und kann vom Absender mit dem abgeschickten Paket verglichen werden. Wenn der Vergleich die kollisionsfreie Übertragung bestätigt, ist der Übertragungsvorgang beendet, und die Übertragungszeit betrug $P + R \cdot P$, die Mindestübertragungszeit. Zeigt der Vergleich aber, daß abgeschicktes und empfangenes Paket sich unterscheiden, lag eine Kollision vor und das Paket muß noch einmal gesendet werden. Wenn das sofort erfolgt, kommt es mit Sicherheit zu einer neuen Kollision, sofern der andere Sender, der an der Kollision beteiligt war, dieselbe Entscheidung trifft. Deswegen wird zufallsmäßig unter den nächsten K Slots einer ausgewählt. Die Verzögerung des Sendens beträgt demnach zwischen 0 und $K - 1$ Slots, jeder von der Länge P und gleichwahrscheinlich. Im Mittel wird darum die Verzögerung $(K-1)/2$ Slots der Länge P betragen. Jede Neuübertragung benötigt dann wieder die Zeit $P + R \cdot P$. Als Zwischenergebnis finden wir für die Gesamtübertragungszeit eines Paketes

$$T = P + R \cdot P + H \cdot \left(\left(\frac{K-1}{2} \right) \cdot P + P + R \cdot P \right)$$

wobei H die Anzahl der Versuche ist, die nötig sind bis das Paket erfolgreich übertragen wurde.

Die Berechnung von H ist einfach, wenn man von einigen Feinheiten absieht. Wie bereits gezeigt wurde, sind bis zur erfolgreichen Übertragung G/S Versuche notwendig. Davon führt der Letzte zum Erfolg. Die vorausgegangenen $G/S - 1$ Versuche waren alle erfolglos. Man kann deswegen $H = G/S - 1$ setzen[4] und findet schließlich mit $G/S = e^G$ beim slotted ALOHA-Verfahren für die Übertragungszeit eines Paketes

$$\frac{T}{P} = 1 + R + (e^G - 1) \cdot \left(\frac{K+1}{2} + R \right)$$

[Kle76, S. 372] gibt für H eine wesentlich kompliziertere und mathematisch wohl auch genauere Gleichung an. Beim Grenzübergang $K \to \infty$ werden die Ausdrücke jedoch identisch.

??? In Bild 4.5 ist die Übertragungsverzögerung beim slotted ALOHA in Abhängigkeit von XXX, für verschiedene Werte von YYY dargestellt.

[3] Sie können sich ohne Benutzung des exklusiv benutzbaren Kanals überhaupt nicht verständigen.

[4] Dies gilt eigentlich nur, wenn $K \to \infty$ geht. Die Anzahl der Versuche ist nur dann G/S, wenn es nicht nur hinreichend viele Sender gibt, sondern auch die erneute Übertragung zu jedem beliebigen Zeitpunkt erfolgen kann.

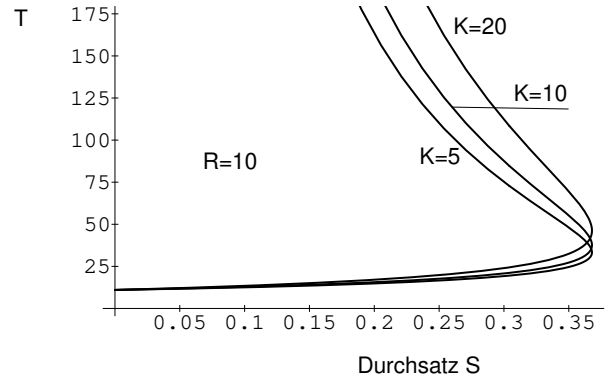

Abbildung 4.5: Verzögerung beim slotted ALOHA.

4.2 CSMA-Protokolle

Die in diesem Kapitel vorgestellten CSMA[5]-Protokolle stellen eine Verfeinerung des bereits behandelten ALOHA-Protokolls dar. Während man bei den ALOHA-Verfahren zu jedem beliebigen Zeitpunkt (pure ALOHA) oder zu getakteten Zeitpunkten (slotted ALOHA) ein Nachrichtenpaket abschickte, in der Hoffnung, daß es korrekt beim Empfänger ankommt, aber auch in der Gewißheit, daß immer wieder Pakete zerstört werden, testet man bei den CSMA-Verfahren zuerst, ob nicht bereits ein anderes Paket gerade auf der Leitung ist. Ist dies der Fall, dann ist es Unsinn, eine Nachricht abzuschicken, da diese mit Sicherheit mit der anderen kollidieren wird.

CSMA-Verfahren können immer angewendet werden, wenn Sender und Empfänger nahe beieinander stehen (wenige Meter bis zu einigen Kilometern), wobei es gleichgültig ist, ob die Übertragung drahtlos oder über Kabel erfolgt. Unter der Annahme, daß es technisch möglich ist, festzustellen, ob auf einer Trägerfrequenz gerade gesendet wird, ist es zweckmäßig, vor dem Senden zunächst in den Übertragungskanal bzw. die Leitung zu horchen. Wenn gerade eine Nachricht übertragen wird, wird das Senden der eigenen Nachricht zurückgestellt.

Bis hierhin sind alle CSMA-Verfahren gleich: man sendet wegen offensichtlicher Erfolglosigkeit nicht. Nun sollte man meinen, daß bei freier Leitung sofort gesendet wird. Aber hier unterscheiden sich die Verfahren. Eine Station kann nämlich nicht feststellen, ob die Leitung wirklich frei ist, sie kann nur feststellen ob sie selbt momentan eine Nachricht auf der Leitung hört oder nicht. Hört sie nichts, kann dennoch ein Paket einer anderen Station, die kurz vorher zu senden begann, bereits unterwegs sein. Deswegen ist es nicht selbstverständlich, bei vermeintlich freiem Kanal sofort zu senden, aus statistischen Überlegungen kann es sinnvoll sein, sich anders zu entscheiden. Auch bei belegtem Kanal gibt es verschiedene Verfahren: Soll man den Kanal bei Gelegenheit nocheinmal testen, oder soll man durch ständiges Mithören den Zeitpunkt abwarten, zu dem der Kanal frei wird?

Diese Überlegungen führen zu mehreren verschiedenen CSMA-Verfahren, die im folgenden vorgestellt werden. Alle Verfahren horchen zunächst in den Übertragungskanal (daher „Carrier Sense“) und senden nicht, wenn der Kanal belegt ist. Von hier an unterscheiden sie sich aber.

4.2.1 Non-persistent CSMA

Diese Verfahren ist—mit Einschränkungen—mit dem Verhalten beim Telefonieren zu vergleichen. Wenn der Apparat des Empfängers frei ist, wird es dort klingeln, d.h. der Kontakt ist hergestellt, was hier der Übertragung der Daten entspricht. Ist der Apparat des Empfängers besetzt, versucht man es irgendwann später nocheinmal. Nonpersistent CSMA verhält sich also nach folgendem Algorithmus:

[5] CSMA = Carrier Sense Multiple Access

1. Wenn der Kanal frei ist, dann wird die Nachricht sofort übertragen.
2. Wenn der Kanal belegt ist, dann wird der Algorithmus nach einem zufallsmäßig bestimmten Zeitintervall nocheinmal gestartet, man beginnt dann wieder bei 1.

Es kann natürlich passieren, daß man einen „falschen“ Zeitpunkt wählt. Hätte man es etwas früher versucht, wäre der Kanal noch frei gewesen, nun ist er schon wieder belegt. Deshalb versucht man mit anderen Protokollen, einen günstigeren Zeitpunkt zu bestimmen.

4.2.2 1-persistent CSMA

Hier ist die sendewillige Station hartnäckig (persistent). Man will auf keinen Fall zulassen, daß sendewillige Stationen nicht senden, obwohl der Kanal frei ist.

1. Wenn der Kanal frei ist, dann wird die Nachricht sofort übertragen.
2. Wenn der Kanal belegt ist, dann horcht die sendewillige Station dauernd in die Leitung und sendet sofort, d.h. mit der Wahrscheinlichkeit 1, wenn der Kanal frei wird.

Kommentar Kleinrock

4.2.3 p-persistent CSMA

Da es nicht möglich ist, den tatsächlichen Zustand der Übertragungsleitung zu ermitteln, ist es vielleicht nicht schlecht, den Zufall ins Spiel zu bringen. Alle Feststellungen und darauf basierende Entscheidungen sind niemals hundertprozentig richtig. Sie treffen nur mit einer nicht näher bekannten Wahrscheinlichkeit zu. Aus diesem Grund, und aus weiteren noch zu erläuternden Überlegungen, verfährt man beim p-persistent CSMA-Protokoll wie folgt:

1. Wenn der Kanal frei ist, dann sendet man mit der Wahrscheinlichkeit p, d.h. man wartet mit der Wahrscheinlichkeit $(1-p)$ eine bestimmte Zeit lang, nämlich τ Zeiteinheiten, wobei τ die maximale Laufzeit zwischen den Stationen ist. (Bei Übertragung per Kabel ist τ die Laufzeit von einem Kabelende zum anderen.) Nach dieser Wartezeit beginnt man wieder von vorne, d.h. testet erneut ob der Kanal frei ist, sendet mit Wahrscheinlichkeit p, usw.
2. Wenn der Kanal belegt ist, dann wartet man auf das Freiwerden der Leitung und beginnt dann wieder bei 1.

Wenn man p gegen Null gehen läßt, kann es sehr lange dauern bis eine Übertragung endlich gestartet wird. Dafür ist man dann aber relativ sicher vor Kollisionen.

4.2.4 Analyse

Es soll nun versucht werden, unter einigen die Rechnung erleichternden Annahmen, eine Analyse der verschiedenen CSMA-Protokolle durchzuführen.

Dabei interessieren besonders der Durchsatz und die Zeit für eine erfolgreiche Übertragung.

Die erste Annahme, die wir bei derartigen Algorithmen immer machen, ist, daß es unendlich viele Stationen gibt, und daß ihr Verhalten durch eine Poisson-Verteilung beschrieben werden kann. Als „Ankünfte“ werden dabei die ankommenden Pakete angesehen, gleichgültig ob sie zum erstenmal oder wegen vorausgegangener Kollision zum wiederholten Male abgesendet werden.

Die *Laufzeit* zwischen je zwei Stationen sei τ, wobei τ ein fester Wert, nämlich die maximale Laufzeit auf der Leitung, ist.

Alle Pakete sollen gleich groß sein. Ihre Größe sei P mit der Dimension *Zeit*. Wir nennen P die *Paketzeit*.

Schließlich nehmen wir noch an, daß die Hardware fehlerfrei arbeitet.

Wir verwenden die gleichen Bezeichner, wie sie bei der Analyse des ALOHA-Systems vorgestellt wurden. G ist die Anzahl der insgesamt (erfolgreich oder nicht) abgeschickten Pakete, bezogen auf P. S bezeichnet den Durchsatz, d.h. die Anzahl der erfolgreich übertragenen Pakete, auch bezogen auf P. Man kann, stationäres Verhalten vorausgesetzt, S auch als Rate ansehen, mit der *neue* Pakete abgeschickt werden, denn unter den genannten Voraussetzungen wird jedes Paket genau einmal erfolgreich übertragen, und nur diese Pakete werden in S gezählt. Daß wegen Kollisionen manche Pakete mehrfach übertragen werden müssen, wird nicht in S, wohl aber in G gezählt.

Analyse des Non-persistent CSMA

Wir gehen von folgender Situation aus: Zu einer beliebigen Zeit t sei die Leitung frei, kein Paket ist unterwegs. Zu diesem Zeitpunkt möge eine Station eine freie Leitung feststellen und ein Paket senden. Wir können nun einige wichtige und charakteristische Zeitpunkte angeben:

t : Die Leitung ist frei, eine Station sendet ein Paket,

$t+\tau$: Da die Laufzeit auf der Leitung τ Zeiteinheiten groß ist und bei t die Leitung frei war, werden alle Stationen bis zu diesem Zeitpunkt glauben, daß die Leitung frei ist und daß sie deswegen senden können. Zur Zeit $(t+\tau)$ kommt bei allen Stationen das erste Bit des zur Zeit t gesendeten Paketes an. Dann wird keine Station mehr senden. Zwischen t und $t+\tau$ können also mehrere Pakete abgeschickt werden.

$t+Y\ (0 \le Y \le \tau)$: Zwischen t und $(t+\tau)$ gibt es einen Zeitpunkt $(t+Y)$, zu dem das letzte Paket im Intervall $[t\,, t+\tau]$ abgeschickt wird. Die Übertragung ist nur dann kollisionsfrei, wenn $Y=0$ ist, d.h. wenn das letzte Paket gleichzeitig das erste und einzige im gefährdeten Zeitintervall τ ist.

$t + Y + \tau + P$: Das letzte Bit des letzten Paketes ist angekommen, danach ist die Leitung frei. Sie war also $B = Y + \tau + P$ Zeiteinheiten lang belegt.

$t_1 \geq t + Y + \tau + P$: Bis zu diesem Zeitpunkt möge die Leitung frei sein, dieses Intervall nennen wir *idle time* I: $I = t_1 - (t + Y + \tau + P)$. Ab dann beginnt das Spiel von vorne.

Die Zykluszeit ist also die Summe $(I + B)$. Wenn in dieser Zeit nur ein Paket übertragen wurde, dann konnte die Leitung optimal genutzt werden. Wir haben den Durchsatz bisher über die Anzahl der Pakete, bezogen auf die Paketzeit P definiert. Danach ist die Auslastung S die für die Übertragung genutzte Zeit, bezogen auf die Zykluszeit, also

$$S = \frac{\overline{U}}{\overline{I} + \overline{B}}$$

wobei $\overline{U}$ die mittlere Zeit für eine erfolgreiche Übertragung ist, und $\overline{I}$ und $\overline{B}$ die Mittelwerte der bereits definierten Größen I und B sind. Da jedes Paket für eine Übertragung, erfolgreich oder nicht, P Zeiteinheiten benötigt, ist

$$\overline{U} = P \cdot \mathcal{P}[\text{das erste Paket im Zyklus wird erfolgreich übertragen}]$$

Das erste Paket wird genau dann erfolgreich übertragen, wenn in der gefährdeten Zeit $[t, t + \tau]$ kein weiteres Paket gesendet wird. Poisson-Verteilung der Ankünfte vorausgesetzt, wird somit

$$\overline{U} = P \cdot e^{-G \cdot \tau / P}$$

Dabei ist G die Ankunftsrate, bezogen auf die Paketzeit P und τ/P das kritische Zeitintervall, ebenfalls bezogen auf P.

Die Belegtzeit B ist eine Zufallsvariable, für die sich der Erwartungswert zu

$$\overline{B} = \overline{Y} + \overline{\tau} + \overline{P}$$

errechnet, wobei $\overline{\tau} = \tau$ und $\overline{P} = P$ ist. Es ist also nur noch der Erwartungswert von Y zu berechnen. Y kann alle möglichen Werte zwischen 0 und τ annehmen. Wenn man die Verteilungsfunktion von Y als $F_Y = \mathcal{P}[Y \leq y]$ definiert, und wenn man sich daran erinnert, daß Y der Sendezeitpunkt des *letzten* Paketes im Intervall $[t, t + \tau]$ ist, dann ist $\mathcal{P}[Y \leq y]$ gleichbedeutend mit der Forderung, daß im Intervall $[t + y, t + \tau]$ ***kein*** weiteres Paket mehr abgesendet wird. Mit der Poisson-Verteilung wird also

$$\begin{aligned} F_Y &= \mathcal{P}[Y \leq y] \\ &= \mathcal{P}[\text{keine Ankünfte in } (\tau - y) \text{ Zeiteinheiten}] \\ &= e^{-G \cdot (\tau - y)/P} \end{aligned}$$

Der Erwartungswert für Y errechnet sich damit zu

$$\overline{Y} = \tau - \frac{P}{G} \cdot \left(1 - e^{-G \cdot \tau / P}\right)$$

und für die Belegtzeit erhält man

$$\overline{B} = P + 2\tau - \frac{P}{G} \cdot \left(1 - e^{-G \cdot \tau / P}\right)$$

Die mittlere Leerzeit (idle time) berechnet sich einfach zu $\overline{I} = P/G$. Die mittlere Zeit zwischen zwei Ankünften ist nämlich beim Poisson-Prozeß P/G, wobei G wieder auf P normiert ist. Dieser Wert gilt wegen der Markov-Eigenschaft (Gedächtnislosigkeit) für jeden beliebigen Beobachtungszeitpunkt, also auch für den oben definierten Zeitpunkt des Beginns der Leerzeit.

Damit sind alle Größen für die Berechnung von S bekannt. Setzt man noch zur Abkürzung $a = \tau/P$, dann wird nach einfacher Rechnung

$$S = \frac{G \cdot e^{-a \cdot G}}{G \cdot (1 + 2a) + e^{-a \cdot G}}$$

Interessant ist noch der Grenzfall $a = 0$ bzw. $\tau = 0$, wenn also jede Station sofort ($\tau = 0$, keine Laufzeit) das Absenden eines Paketes einer anderen Station erkennt. Es wird

$$\lim_{\tau \to 0} S = \frac{G}{G + 1}$$

Bild 4.6 zeigt den Durchsatz beim Non-persistent CSMA für verschiedene Werte von a.

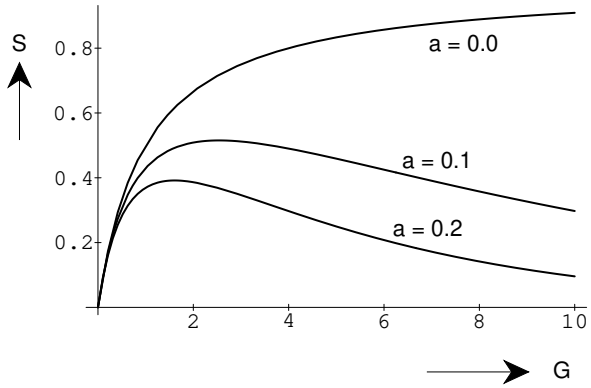

Abbildung 4.6: Durchsatz beim Non-persistent CSMA

5. Das Problem des gegenseitigen Ausschlusses

5.1 Einführung

Alle Rechenanlagen arbeiten, ohne daß der Benutzer davon bewußt Kenntnis nimmt, in vielfacher Weise intern parallel. Insbesondere ist auf der Ebene der Hardware Parallelität an vielen Stellen festzustellen. So können z.B. mehrere Ein-/Ausgabegeräte gleichzeitig arbeiten. Auch die Tätigkeiten des Auslesens von Befehlen aus dem Arbeitsspeicher und deren Entschlüsselung laufen oft zeitlich überlappt ab. Man sollte jedoch den Begriff der Parallelität nicht zu eng fassen. Auch wenn die Anlage nur über einen einzigen Rechnerkern (CPU = central processing unit) verfügt, können mehrere Benutzer-Prozesse parallelärbeiten.

Wenn zwei oder mehrere Statements (oder Statementfolgen) im erweiterten Sinn *parallel* ablaufen, so wollen wir darunter verstehen, daß sie in beliebiger, nicht festlegbarer Reihenfolge, die dann im Grenzfall sequentiell sein kann, ablaufen dürfen. Das soll in

```
s1; PARBEGIN s2; s3; s4 PAREND; s5
```

zum Ausdruck kommen. Nach Ausführung des Statements $s1$ dürfen die Statements $s2$, $s3$ und $s4$ parallel bearbeitet werden. Sie können also nach obiger Festlegung in beliebiger Reihenfolge ausgeführt werden. Das Statement $s5$ wird jedoch erst dann bearbeitet, wenn $s2$, $s3$ und $s4$ fertig geworden sind. Dazu ein Beispiel:

```
read( k );
PARBEGIN
   FOR i:=1 TO k DO a[i]:=0;
   FOR j:=1 TO k DO b[j]:=1;
   read (z);
PAREND;
a[z]:=z;
```

Es werden die beiden Felder a und b von 1 bis k initialisiert. Zuvor wird k eingelesen. Da beide Felder und ihre Initialisierung unabhängig voneinander sind, darf die Initialisierung gleichzeitig (parallel) ablaufen. Angenommen, der Rechner verfügt über zwei oder mehr Zentraleinheiten, dann könnten beide Felder *echt parallel* besetzt werden. Außerdem läuft „gleichzeitig“ das Einlesen einer Größe z. Erst wenn beide Felder voll besetzt sind und z eingelesen worden ist, wird `a[z]:=z` ausgeführt.

Generell darf die Reihenfolge der Bearbeitung zwischen PARBEGIN und PAREND beliebig sein. Vielleicht wird sie wie angegeben vorgenommen, vielleicht wird aber auch wechselweise ein a[i] besetzt und dann ein b[j] und read(z) wird vorgezogen. Wir dürfen also bei der Behandlung

paralleler Prozesse keine Annahme über die Reihenfolge und auch keine Annahme über die Bearbeitungsgeschwindigkeit machen. Wir dürfen nicht einmal annehmen, daß bei einem zweiten Durchlauf des Programms die gleiche Reihenfolge wie beim ersten Durchlauf eingehalten wird. Die Reihenfolge ist vielmehr nicht reproduzierbar, wodurch Programmtests zumindest erschwert, wenn nicht gar unmöglich gemacht werden.

Wir haben bis jetzt zur Darstellung paralleler Abläufe die Schreibweise einer Programmiersprache verwendet. Üblich ist aber auch eine Beschreibung durch einen Präzedenzgraphen, der für

```
s1; PARBEGIN s2; s3; s4 PAREND; s5
```

wie in Bild 5.1 aussieht.

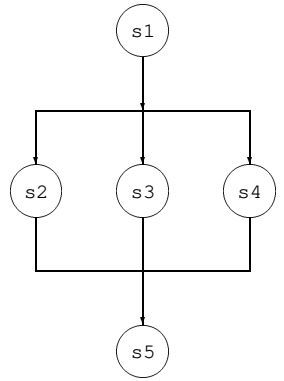

Abbildung 5.1: Präzedenzgraph

Solche Graphen können natürlich viel komplizierter aussehen. Um sie zu erzeugen, muß man wissen, in welcher zeitlichen Reihenfolge die Aktionen (oben durch Statements ausgedrückt) auszuführen sind. Wir wollen dazu eine Ordnungsrelation „$<$“ definieren, die bedeuten soll, daß bei $s_a < s_b$ die Aktion s_a vor der Aktion s_b auszuführen ist. In unserem Beispiel ist:

$$s_1 < s_2 \quad s_1 < s_3 \quad s_1 < s_4$$
$$s_2 < s_5 \quad s_3 < s_5 \quad s_4 < s_5$$

Wir wollen für „$<$“ folgende Eigenschaften festlegen:

irreflexiv : $s_a \neq s_a$

transitiv : $s_a < s_b \wedge s_b < s_c \Longrightarrow s_a < s_c$

asymmetrisch : $s_a < s_b \Longrightarrow s_b \neq s_a$

Die Aufgabe, aus gegebenen Relationen $s_a < s_b$ den Präzedenzgraphen zu erstellen, ist keine spezielle Aufgabe der Informatik. Das gleiche Problem tritt z.B. in der Netzplantechnik des Operations Research auf und kann mit Algorithmen zum topologischen Sortieren gelöst werden ([KNU75]).

Es gelingt übrigens nur dann, einen Präzedenzgraphen zu erstellen, wenn es nicht möglich ist, aus den gegebenen Relationen die Kette

$$s_a < s_b < \ldots < s_a$$

zu bilden. Man kann zwar auch dann einen gerichteten Graphen aufbauen, aber er ist nicht zyklenfrei, d.h. es liegt ein Deadlock vor, da die Aktion s_a schließlich auf sich selbt wartet.

Es wird in Zukunft eine wichtige Aufgabe bei der Konstruktion von Hardware und Software sein, mögliche Parallelitäten auszunutzen. Im Sortierverfahren Quicksort können z.B. zahlreiche Operationen parallel ausgeführt werden ([Hoa72]).

Wir werden in diesem Kapitel mehrere Methoden kennenlernen, um den gegenseitigen Ausschluß zu garantieren. Diese Methoden sind z.T. Hardwarenah und aus diesem Grund in der Regel nur für privilegierte Prozesse erlaubt. Sie müssen unter voller Kontrolle des Betriebssystems ablaufen. Außerdem müssen sie unbedingt auf die gegebenen Hardwaremöglichkeiten Rücksicht nehmen. Wir wollen zunächst die folgenden beiden Modelle verwenden, wie sie in Bild 5.2 dargestellt sind.

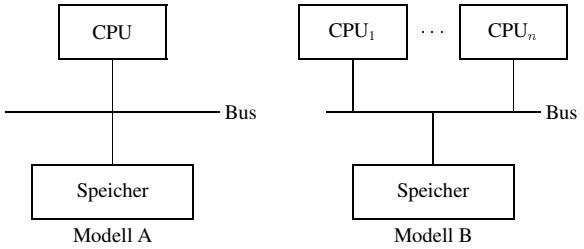

Abbildung 5.2: Modell eines Rechners mit einer und mit mehreren CPUs

Im ersten Modell gibt es nur eine einzige CPU, die über einen Bus auf den Speicher (RAM) zugreift. Im zweiten Modell gibt es mehrere CPUs, die über einen Bus auf einen gemeinsamen Speicher zugreifen. Die CPUs können zwar unabhängig voneinander arbeiten, aber beim Zugriff auf den Speicher konkurrieren sie miteinander. Die Hardware wird eine — nicht deterministische — Reihenfolge erzeugen, da immer nur ein Prozessor auf den Speicher lesend oder schreiben zugreifen kann.

Das Problem des gegenseitigen Ausschlusses kann (muß) für die beiden Modelle unterschiedlich gelöst werden. Mit einem weiteren Modell werden wir uns erst später befassen. Es besteht aud mehreren Systemen, die aus Modell A oder Modell B oder aus beiden gemischt bestehen können, und die durch Nachrichtenaustausch miteinander kommunizieren können. Wegen der endlichen Ausbreitungsgeschwindigkeit der Nachrichten müssen sie irgendwie mit den Laufzeiten der Nachrichten fertig werden.

Die Technik des gegenseitigen Ausschlusses kann an einem einfachen Beispiel gezeigt werden. In zwei parallelen Prozessen möge jeweils die Anweisung `x:=x+1` vorkommen. Daraus wird in der Regel die Maschinenbefehlsfolge

```
lade x in ein Register
addiere 1
schreibe Ergebnis zurück
```

Man überzeugt sich leicht, daß wegen der Nichtdeterminiertheit des Ablaufes der Prozesse nach der Addition von 1, die in beiden Prozessen erfolgt, x nicht unbedingt um 2 größer geworden ist, wie es sein sollte. Wenn nämlich ein Prozeß die Addition beginnt, bevor der (zeitlich) erste Prozeß das Ergebnis zurückgeschrieben hat, dann verwenden beide Prozesse den gleichen alten x-Wert für die Addition. Dann wird x insgesamt nur um 1 erhöht, obwohl beide Prozesse jeweils eine 1 addiert haben. Man muß also dafür sorgen, daß in diesem Beispiel die Addition mit anschließendem Zurückschreiben ungestört verlaufen kann. Wir werden Synchronisierungsmethoden kennenlernen, die auch in weitaus komplizierteren Fällen anwendbar sind. Für den hier gezeigten, besonders einfachen Fall, gibt es Möglichkeiten, ohne großen Aufwand das gewünschte Ergebnis zu erzielen. Das gezeigte Beispiel ist deswegen so einfach, weil die Befehlsfolge, die, um richtig zu funktionieren, unteilbar sein sollte, nur elementare Operationen enthält, die für nur wenige Mikro-Sekunden die CPU belegen. In diesem Fall genügt es, die CPU zu zwingen, alle an der Ausführung beteiligten Befehle hintereinander ohne Unterbrechung zu bearbeiten. Wenn die Anlage nur eine einzige CPU besitzt, brauchen nur die Interrupts gesperrt zu werden, denn nur durch Interrupts kann es hier zu einem Wechsel in der Prozeßbearbeitung kommen:

```
sperre Interrupts;
x:=x+1;
gib Interrupts frei;
```

Die Befehle, die die Interrupts steuern, sind privilegiert, können also von „normalen“ Benutzern nicht verwendet werden. Diese hätten nämlich sonst die Möglichkeit, ohne vom Betriebssystem daran gehindert zu werden, den Rechner für beliebig lange Zeit ausschließlich für sich zu reservieren. Aber auch der Programmierer von Betriebssystemkomponenten muß vorsichtig mit dem Sperren von Interrupts umgehen. Diese Methode sollte nur dann verwendet werden, wenn zwischen dem Sperren und dem Freigeben der Interrupts nur wenige einfache Operationen ausgeführt werden.

Andernfalls können die während dieser Zeit erzeugten aber noch nicht behandelten Interrupts verloren gehen. Wenn die Anlage nur einen einzigen Prozessor besitzt, ist diese Methode dann allerdings anderen, aufwendigeren Methoden vorzuziehen. Sie versagt jedoch, wenn die Anlage über mehrere Prozessoren verfügt, die auf gemeinsame Daten im gemeinsamen Speicher zugreifen können. Man kann nämlich immer nur die Interrupts des eigenen Prozessors sperren, hat aber keinen Einfluß auf den Interruptmechanismus der anderen Prozessoren. Deswegen verfügen insbesondere Mehrprozessoranlagen (aber auch Monoprozessoranlagen) über `LOCK`- und `UNLOCK`-Befehle, die nach folgendem Schema arbeiten [KEED79], [DENN71]:

- `lock(x)` entspricht den Befehlen

  ```
  LOCK: { IF x=0 THEN x:=1 ELSE GOTO LOCK };
  ```

- `unlock(x)` entspricht den Befehlen

  ```
  UNLOCK: { x:=0 };
  ```

Dabei soll x eine Variable sein, auf die alle Prozessoren zugreifen können. Die Konstruktion „$\{ \ldots \}$“ bedeutet, daß die darin enthaltenen Anweisungen nicht unterbrechbar sind, also als unteilbare Operation gelten.

Die Variable x ist mit Null vorbesetzt. Die Operation `lock(x)` testet die Variable x solange bis diese den Wert Null annimmt. In diesem Fall wird x auf Eins gesetzt und der Prozeß betritt das „kritische Gebiet“. Ein *kritisches Gebiet* ist dadurch ausgezeichnet, daß die in ihm liegenden Operationen nur von einem einzigen Prozeß ausgeführt werden dürfen, daß aber ohne Synchronisierungsmaßnahmen mehrere Prozesse gleichzeitig darauf arbeiten könnten. Solange ein Prozeß im kritischen Gebiet ist, wird jeder andere Prozeß, der `lock(x)` ausführt, in der Warteschleife bleiben bis durch `unlock(x)` die Variable x auf Null gesetzt wird. Dann kann einer der wartenden Prozesse, und wegen der Unteilbarkeit der Abfrage genau einer, die Variable wieder auf Eins setzen und das kritische Gebiet betreten. Die Programmierung ist jetzt sehr einfach geworden:

Prozeß$_i$	Prozeß$_j$
...	...
lock(x);	lock(x);
kritisches Gebiet	kritisches Gebiet
unlock(x);	unlock(x);

Obwohl damit eine korrekte Lösung des gegenseitigen Ausschlusses gegeben ist, gibt es einige Punkte zu beachten. Wenn ein Prozeß auf ein gesperrtes kritisches Gebiet trifft, dann verbringt er die weitere Zeit ausschließlich mit der Abfrage auf das Freiwerden des kritischen Gebietes (*busy waiting/polling*). Der Prozessor wird also mit der ständigen Abfrage beschäftigt, auch wenn er inzwischen andere Arbeiten erledigen könnte. Besser wäre es, wenn der abfragende Prozeß still gelegt würde und genau dann wieder aktiviert würde, wenn er das kritische Gebiet betreten kann.

Schlimmer ist, daß die angegebene Lösung nicht sicher ist und zum Deadlock führen kann. Nehmen wir an, daß in einem Prozessor die Prozesse nach Prioritäten geordnet sind und die Prozessorvergabe den Prioritäten entsprechend erfolgt.

> Priorität $\Longrightarrow$:
> Prozesse: A B C . . . D E

Nun möge ein Prozeß niedriger Priorität (z.B. Prozeß B) im kritischen Gebiet sein. In dieser Zeit versucht ein Prozeß höherer Priorität (z.B. Prozeß D) dasselbe kritische Gebiet zu betreten und führt `lock(x)` aus. (Er kann das, wenn der Prozessor Gelegenheit bekommt, nach Ablauf einer Zeitscheibe, d.h. nach Ablauf einer Zeit von einigen 100 ms, eine neue Verplanung durchzuführen.) Er wird ständig die Variable x abfragen, die aber ihren Wert nicht ändert, da derjenige Prozeß, der allein x ändern kann, wegen seiner niedrigen Priorität nicht mehr in den Besitz des Prozessors kommt. Der abfragende Prozeß mit der höheren Priorität „verdeckt“ ihn. So wartet der eine Prozeß darauf, daß das kritische Gebiet frei wird, und der andere wartet darauf, den Prozessor zu erhalten, um das kritische Gebiet freizugeben. Beide warten also aufeinander, die typische Deadlocksituation. Sie kann durch dynamische Prioritäten aufgehoben werden. Wenn ein Prozeß eine bestimmte Zeit vergeblich in `lock(x)` gewartet hat, wird seine Priorität verschlechtert. Er wird schließlich so wenig wichtig werden, daß seine Priorität geringer wird als die des im kritischen Gebiet befindlichen Prozesses, der dann endlich die Gelegenheit erhält, `unlock(x)` auszuführen, falls nicht inzwischen ein anderer Prozeß in `lock(x)` rotiert, wodurch das Spiel von vorne beginnt.

Auf Anlagen der Rechnerserie Siemens 7.000 gibt es für `lock(x)` den Maschinenbefehl „`TS x`“, wobei x die Adresse eines Byte ist. (TS=test and set). Der Befehl TS ist unteilbar und arbeitet auch im Mehrprozessorbetrieb exklusiv auf der Adresse x. Er vergleicht den Inhalt von x mit Null, hält das Ergebnis in der Anzeige fest und setzt das Byte auf Adresse x in allen Bits auf Eins. Die Konstruktion

```
lock(x);
kritisches Gebiet;
unlock(x);
```

kann damit auf Siemens-Anlagen wie folgt programmiert werden:

```
X DC X'00'          x mit 0 vorbesetzen
...
LOOP TS X           test and set
BZ LOOP             lock(x)
   kritisches Gebiet
MVI X,0             unlock(x)
```

??? Hinweis auf MC68XXX, Intel XX86.

5.2 Synchronisierung nach Dekker ([Dij68])

Ohne Einschränkung der Allgemeinheit wollen wir nur zyklische, parallel arbeitende Prozesse betrachten, die sog. kritische Abschnitte durchlaufen. Ein kritischer Abschnitt war z.B. im vorigen Kapitel die Anweisung `x:=x+1`, weil die Möglichkeit bestand, daß mehrere Prozesse gleichzeitig auf die Variable x, die „gemeinsame Variable“ zugriffen.

Damit derartige Operationen korrekt durchgeführt werden, müssen wir verlangen, daß sie in einem kritischen Gebiet liegen und durch entsprechende Maßnahmen vor einem Mehrfachzugriff geschützt sind. Solange wir keine Synchronisierungsoperationen kennen und das Sperren von Interrupts nicht oder nur in Ausnahmefällen zulassen müssen wir mit den einzigen uns bekannten unteilbaren Operationen (nur unteilbare Operationen können uns nach unserem derzeitigen Wissensstand Exklusivität garantieren) „Lesezugriff auf den Speicher“ und „Schreibzugriff auf den Speicher“ leben. Um die Anweisung `x:=x+1` wie eine unteilbare Operation (hier: wie ein kritisches Gebiet) zu behandeln, liegt folgende Lösung nahe, die in allen Prozessen, die `x:=x+1` ausführen wollen, durchlaufen werden muß:

```
WHILE NOT free DO;
free:=false;
x:=x+1;
free:=true;
```

(Man beachte hier und in den folgenden Beispielen die leere Anweisung). Dabei ist *free* als globale Größe angenommen, die mit `free:=true` initialisiert worden sein soll. Der angegebene Lösungsvorschlag ist sicher nicht korrekt, denn die Abfrage von free in der while-Schleife müßte auch in einem kritischen Gebiet liegen. Jetzt ist es nämlich möglich, daß mehrere Prozesse gleichzeitig die Variable *free* abfragen und gleichzeitig feststellen, daß *free=true* ist und damit im Gegensatz zur Forderung, daß immer nur höchstens ein Prozeß im kritischen Gebiet ist, das kritische Gebiet betreten. Wir müssen also nach anderen Lösungen suchen. Dabei sollen die zyklischen Prozesse durch das Wortsymbol `CYCLE` gekennzeichnet sein. Ein zyklischer Prozeß hat folgenden Aufbau:

```
process i: CYCLE
BEGIN
   tue 1;
   kritisches Gebiet;
   tue 2
END.
```

„tue1“ und „tue2“ sind Anweisungen, die nicht im kritischen Gebiet liegen müssen und für die Synchronisierungsaufgabe ohne Bedeutung sind. Wir wollen zunächst annehmen, daß wir es nur mit zwei zyklischen Prozessen zu tun haben.

```
VAR turn: integer;
```



```
BEGIN
   turn:=1;

   PARBEGIN

      process 1: CYCLE
      BEGIN
         WHILE turn = 2 DO;
         critical section 1;
         turn:=2;
         remainder of cycle 1
      END;

      process 2: CYCLE
      BEGIN
         WHILE turn = 1 DO;
         critical section 2;
         turn:=1;
         remainder of cycle 2
      END

   PAREND
END.
```

Da vor Beginn der parallelen Prozesse *turn* den Wert 1 hat, kann nur der Prozeß 1 das kritische Gebiet betreten, während der zweite Prozeß, falls er das kritische Gebiet betreten will, vorher in der while-do-Schleife auf der Stelle tritt. Erst nach Verlassen des kritischen Gebietes wird Prozeß 1 *turn* auf 2 setzen. Erst dann kann Prozeß 2 in das kritische Gebiet gelangen. Damit scheint die Synchronisierung gelungen zu sein. Sie ist aber zu restriktiv, denn hier wurde unnötigerweise, ohne es zu wollen, gefordert, daß die beiden Prozesse immer nur abwechselnd das kritische Gebiet betreten. Wäre z.B. Prozeß 2 erheblich schneller als Prozeß 1, so müßte er doch immer darauf warten, daß dieser die Variable *turn* umschaltet. Prozeß 2 würde also unnötigerweise gebremst. Er würde sogar für immer angehalten, wenn Prozeß 1 den Zyklus verlassen hat (das kommt im Programm nicht explizit vor) oder absichtlich oder durch einen Defekt außerhalb des kritischen Gebietes stehengeblieben ist. Wir müssen deswegen in unsere weiteren Lösungsversuche auch noch die Bedingung einbauen, daß, wenn der eine Prozeß außerhalb des kritischen Gebietes stehenbleibt, der andere Prozeß dennoch weiterarbeiten kann.

```
VAR s1, s2: integer;
BEGIN
   s1:=1; s2:=1;

   PARBEGIN

      process 1: CYCLE
```

```
      BEGIN
         WHILE s2 = 0 DO;
         s1:=0;
         critical section 1;
         s1:=1;
         remainder of cycle 1
      END;

      process 2: CYCLE
      BEGIN
         WHILE s1 = 0 DO;
         s2:=0;
         critical section 2;
         s2:=1;
         remainder of cycle 2
      END

   PAREND
END.
```

Hier ist das erwünschte Ziel aus bereits angegebenen Gründen nicht erreicht. Die Exklusivität der kritischen Bereiche ist nämlich nicht gewährleistet. Die beiden Größen $s1$ und $s2$ sollen anzeigen, ob sich der jeweilige Prozeß im kritischen Bereich befindet. Aber beide Größen können hier unabhängig voneinander, also auch gleichzeitig geändert werden, so daß beide Prozesse das kritische Gebiet betreten können. Wie bereits gesagt, muß man den Test der Synchronisationsvariablen $s1$ und $s2$ deswegen in das kritische Gebiet mit einbeziehen. Das ist leicht durch eine kleine Änderung des letzten Programms möglich.

```
VAR s1, s2 : integer;
BEGIN
   s1:=1; s2:=1;

   PARBEGIN

      process 1: CYCLE
      BEGIN
         s1:=0;
         WHILE s2 = 0 DO;
         critical section 1;
         s1:=1;
         remainder of cycle 1
      END;

      process 2: CYCLE
      BEGIN
         s2:=0;
         WHILE s1 = 0 DO;
```



```
         critical section 2;
         s2:=1;
         remainder of cycle 2
      END;

   PAREND
END.
```

Bei dieser Variante ist der gegenseitige Ausschluß in jedem Fall garantiert. Allerdings wurde die Sicherheit zu weit getrieben, weil $s1$ und $s2$ zu früh auf Null gesetzt werden. Denn jetzt kann es passieren, daß beide Prozesse gleichzeitig $s1$ bzw. $s2$ auf Null setzen. Dann warten sie bis in alle Ewigkeit aufeinander, weil niemand mehr $s1$ oder $s2$ auf Eins setzt. Das ist aber die Bedingung zum Betreten eines kritischen Gebiets. Man kann die zu strenge Abschirmung des kritischen Bereiches auflockern, indem jeder Prozeß in der while-Schleife für eine kurze Zeit $s1$ bzw. $s2$ wieder auf 1 setzt, um dem anderen Prozeß die Möglichkeit zu geben fortzufahren.

```
VAR s1, s2: integer;
BEGIN
   s1:=1; s2:=1;

PARBEGIN

   process 1: CYCLE
   BEGIN
      s1:=0;
      WHILE s2 = 0 DO
      BEGIN s1:=1; s1:=0 END;
      critical section 1
      s1:=1;
      remainder of cycle 1
   END;

   process 2: CYCLE
   BEGIN
      s2:=0;
      WHILE s1 = 0 DO
      BEGIN s2:=1; s2:=0 END;
      critical section 2;
      s2:=1;
      remainder of cycle 2
   END;

   PAREND
END.
```

Auch dieser Versuch ist nicht die gesuchte Lösung, weil er gegen die Forderung der Zeitinvarianz verstößt. Die beiden Prozesse verhalten sich

wie zwei miteinander sprechende Fernsprechteilnehmer, deren Gespräch zusammenbricht und die nun versuchen, sich gegenseitig wieder anzurufen. Sie werden dabei fast immer den Apparat des anderen besetzt finden. Im allgemeinen wird dennoch nach endlich vielen Versuchen wieder eine Verbindung zustande kommen.

Der hier gemachte Versuch ist nicht so unsinnig wie man etwa auf Grund der Konstruktion „`s1:=1; s1:=0`"glauben könnte. Tatsächlich muß diese Version mitunter in der Praxis verwendet werden. So können z.B. weit voneinander entfernte Rechner über einen (stationären) Satelliten miteinander kommunizieren. Sie verwenden dazu oft völlig unkoordiniert denselben Übertragungskanal. Eine Nachricht kann also nur dann korrekt übertragen werden, wenn während der Übertragungszeit kein anderer Rechner auf demselben Kanal eine Nachricht abschickt. Der Absender kann aber leicht kontrollieren, ob seine Nachricht korrekt übertragen wurde, denn er kann ja die vom Satelliten übertragene Nachricht genauso wie der Empfänger mithören. Im Fehlerfall muß er noch einmal versuchen, die Nachricht zu übertragen. Das tut er solange bis die Übertragung geglückt ist ([Kle76]. Diese Versuche werden nie zum Erfolg führen, wenn alle Absender zur gleichen Zeit feststellen, daß ihre Nachrichten zerstört wurden und zur gleichen Zeit eine erneute Übertragung starten. Deswegen beginnt jeder seine Übertragung erst nach einer zeitlichen Verzögerung, deren Größe durch eine Zufallszahl ermittelt wird, um damit anderen Absendern eine Chance zu geben. Die Konstruktion

```
BEGIN
   s1:=1;
   s1:=0;
END;
```

bedeutet also sinngemäß:

```
BEGIN
   etwas warten und dem anderen
   Prozess eine Chance geben
END;
```

Nach soviel vergeblichen Versuchen liegt die Annahme nahe, daß eine korrekte Lösung unmöglich ist. Es gibt jedoch eine korrekte Lösung, die auf Decker zurückgeht und Elemente der dargestellten Versuche enthält [Dij68]:

```
VAR s1, s2, turn: integer;
BEGIN
   s1:=1; s2:=1; turn:=1;

   PARBEGIN

      process 1: CYCLE
      BEGIN
         k1: s1:=0;
```



```
         l1: IF s2 = 0 THEN BEGIN
            IF turn = 1 THEN GOTO l1;
            s1:=1;
            WHILE turn = 2 DO;
            GOTO k1
         END;
         critical section 1;
         turn:=2; s1:=1
         remainder of cycle 1
      END;

      process 2: CYCLE
      BEGIN
         k2: s2:=0;
         l2: IF s1 = 0 THEN  BEGIN
            IF turn = 2 THEN GOTO l2;
            s2:=2;
            WHILE turn = 1 DO;
            GOTO k2
         END;
         critical section 2;
         turn:=1; s2:=2
         remainder of cycle 2
      END;

   PAREND
END.
```

Für Anhänger des goto-losen Programmierens kann man die gefundene Lösung noch umschreiben: Diese Umformung ist grundsätzlich möglich aber nicht immer ganz einfach. Der Leser möge, bevor er weiterliest, die Umformung einmal selbst versuchen.

```
VAR s1, s2, turn: integer;
BEGIN
   s1:=1; s2:=1; turn:=1;

   PARBEGIN

      process 1: CYCLE
      BEGIN s1:=0;
         WHILE s2 = 0 DO
         IF turn = 2 THEN BEGIN
            s1:=1;
            WHILE turn = 2 DO;
            s1:=0
         END;
         critical section 1;
         turn:=2; s1:=1;
```

```
      remainder of cycle 1
   END;

   process 2: CYCLE
   BEGIN s2:=0;
      WHILE s1 = 0 DO
      IF turn = 1 THEN BEGIN
         s2:=1;
         WHILE turn = 1 DO;
         s2:=0
      END;
      critical section 2;
      turn:=1; s2:=1;
      remainder of cycle 2
   END

PAREND
END.
```

Natürlich läßt sich auch noch die `CYCLE`-Konstruktion durch eine `WHILE TRUE DO ...`-Schleife oder `REPEAT ... UNTIL FALSE`-Schleife ersetzen...

5.3 Synchronisation von N Prozessen

Wir wollen die im letzten Kapitel für zwei Prozesse gelöste Aufgabenstellung nun auf N ($N > 2$) Prozesse erweitern. Diese N Prozesse können über einen gemeinsamen Speicher miteinander kommunizieren. Lese- und Schreibzugriffe auf den Speicher sind unteilbare Operationen, d.h. wenn mehrere Prozesse gleichzeitig auf denselben Speicherplatz zugreifen wollen, dann werden die Zugriffe einer nach dem anderen in nicht bekannter und nicht voraussehbarer Reihenfolge ausgeführt.

Die Lösung muß, wie schon beim Fall $N = 2$, folgende Bedingungen erfüllen:

1. die Lösung soll bezüglich aller N Prozesse symmetrisch sein.
2. die Arbeitsgeschwindigkeit der einzelnen Prozesse sei unbekannt, beliebig und darf sich jederzeit ändern.
3. wenn ein Prozeß außerhalb seines kritischen Gebiets anhält, so müssen dennoch die anderen Prozeße in der Lage sein, ihre Arbeit fortzusetzen, d.h. der Ausfall eines einzelnen Prozesses soll nicht zum Ausfall des gesamten Systems führen.
4. wenn mehrere Prozesse das kritische Gebiet betreten wollen, dann soll genau einer von ihnen dazu ausgewählt werden. Es darf also nicht vorkommen, daß mehr als ein Prozeß im kritischen Gebiet ist, und es muß ebenso unmöglich sein, daß auf Grund mangelnder Entscheidungsfreudigkeit (bitte nach Ihnen – bitte nach Ihnen)

niemand in das freie kritische Gebiet gelassen wird.

Eine Lösung, die die genannten Forderungen erfüllt, hat zum erstenmal Dijkstra ([Dij65]) angegeben. Der gemeinsame Speicher, auf den alle Prozesse zugreifen können, besteht aus zwei booleschen Feldern und einer Integer-Größe:

```
Boolean array b, c [1:N]; integer k;
```

(Das Programm wird zunächst wie bei Dijkstra in ALGOL 60 angegeben). Wir nehmen an, daß beide Felder mit `true` vorbesetzt sind. Der Initialwert von k ist beliebig, er muß nur zwischen 1 und N liegen. Für den Prozeß i ($1 \leq i \leq N$) gilt das Programm:

```
INTEGER j;
Li0:  b[i]:=FALSE;
Li1:  IF k <> i THEN
Li2:  BEGIN c[i]:=TRUE;
Li3:     IF b[k] THEN k:=i;
   GOTO Li1
END
ELSE
Li4:  BEGIN c[i]:=FALSE;
  FOR j:=1 STEP 1 UNTIL N DO
    IF j <> i AND NOT c[j] THEN GOTO Li1
END;
critical section;
c[i]:=TRUE; b[i]:=TRUE;
remainder of the cycle in which stopping is allowed;
GOTO Li0
```

Wir sehen, daß die Größe k von jedem Prozeß geändert werden kann, während die Feldelemente $b[i]$ und $c[i]$ nur von Prozeß i, dem Eigentümer dieser Feldelemente, geändert werden. Auf die anderen Feldelemente greift Prozeß i nur lesend zu.

Das Feld b hat folgende Bedeutung:

`b[i]=false` : der Prozeß i bewirbt sich um den Eintritt in das kritische Gebiet oder ist im kritischen Gebiet.

`b[i]=true` : sonst.

Die Bedeutung des Feldes c ist schwieriger zu erkennen. Es ist offensichtlich für den Prozeß i nur dann von Bedeutung, wenn gleichzeitig `b[i]=false` ist.

`NOT b[i] AND c[i]` : der Prozeß i versucht, sich selbst in den engeren Kreis der Bewerber zu bringen, d.h. die Größe k, deren Wert den entsprechenden Prozeß zum Eintritt in das kritische Gebiet befähigt, auf i zu setzen.

`NOT b[i] AND NOT c[i]` : dem Prozeß i ist es gelungen, k den Wert i zu geben.

Bevor wir zeigen, daß das Programm alle gestellten Aufgaben erfüllt, soll es zunächst in ein vollständiges, goto-freies Pascal-Programm umgeformt werden. Statt der booleschen Felder wollen wir dabei sinngemäß Mengen verwenden. (In ALGOL 60 gibt es keine Mengen.)

Zunächst ein wichtiger Hinweis: Wir müssen bezgl. der Implementierung annehmen, daß beim Zugriff auf das boolesche Array $b[i]$ nur dieses eine Feldelement gezielt angesprochen wird. Dasselbe gilt für den Zugriff auf ein Mengenelement. Die Siemens-Implementierung von Mengen kann bei `c:=c-[i]` zu falschen Ergebnissen führen! Wegen der Byteadressierung wird nämlich nicht nur ein einziges Bit angesprochen sondern auch die 7 Bit breite Umgebung.

Bsp.: $b[i]$ hat die Größe eines Byte. Dann kann $b[i]$ mit einem Byte-Befehl geändert werden. b(Menge) belegt pro Element ein Bit. Dann darf eine Änderung nur über einen Bit-Befehl erfolgen.

Die Initialisierung in folgendem Programm ist:

```
b:=[1..N];
c:=[1..N];
k:=beliebig; (* 1<=k<=N *)

(*  1 *)  PROGRAM process_i (b, c, k);
(*  2 *)  CONST i = ...;      (* eigene Prozessnummer *)
(*  3 *)        N = ...;      (* Anzahl der Prozesse *)
(*  4 *)  TYPE  T = 1..N;
(*  5 *)  VAR y, b, c: SET OF T;
(*  6 *)            k: T;
(*  7 *)           ok: boolean;
(*  8 *)  BEGIN
(*  9 *)  WHILE true DO BEGIN
(* 10 *)    ok:=false;
(* 11 *)    b:=b-[i];   y:=[1..N]-[i];
(* 12 *)    REPEAT
(* 13 *)      IF k <> i THEN BEGIN
(* 14 *)         c:=c+[i];
(* 15 *)         IF k IN b THEN k:=i
(* 16 *)      END
(* 17 *)      ELSE BEGIN
(* 18 *)         c:=c-[i];
(* 19 *)         ok:=(c+[i])=[1..N]
(* 20 *)      END;
(* 21 *)    UNTIL ok;
(* 22 *)    critical section;
(* 23 *)    c:=c+[i];
(* 24 *)    b:=b+[i];
(* 25 *)    remainder of cycle
(* 26 *)    END   (* while true *)
(* 27 *)  END.
```


Es soll zunächst gezeigt werden, daß es nicht möglich ist, daß zwei Prozesse gleichzeitig in ihrem kritischen Gebiet sind. Die einzige Möglichkeit, das kritische Gebiet zu betreten, ist das Verlassen der repeat-Schleife (Zeilen 12 bis 21). Sie kann nur verlassen werden, wenn der betreffende Prozeß als einziger nicht in der Menge c ist (Zeile 19).

Solange er im kritischen Gebiet ist, ist er nicht in der Menge c enthalten, deswegen kann kein anderer Prozeß feststellen, daß er selbst als einziger nicht in der Menge c ist.

Es muß weiter gezeigt werden, daß es nicht möglich ist, daß wegen mangelnder Entscheidungsfreudigkeit niemand das kritische Gebiet betritt.

Wenn der k-te Prozeß nicht unter den abfragenden Prozessen ist, dann ist er in der Menge b und jeder abfragende Prozeß findet $k \neq i$.

Einer von ihnen oder mehrere finden $k \in b$ und setzen `k:=i`. Danach ist immer $k \notin b$, d.h. k kann nicht mehr verändert werden und bezeichnet jetzt einen derjenigen Prozesse, die das kritische Gebiet betreten wollen.

Es ist jetzt immer noch möglich, daß mehrere Prozesse gleichzeitig den else-Zweig (Zeilen 17 bis 20) betreten. Höchstens einer von ihnen (eventuell also auch keiner) kann feststellen, daß er als einziger nicht in der Menge c ist. Selbst wenn jetzt alle den Test erfolglos beenden, findet genau einer von ihnen `k=i` (in Zeile 13) und kann anschließend das kritische Gebiet betreten.

Die von Dijkstra gefundene Lösung ist jedoch unbefriedigend, weil es möglich ist, daß ein Prozeß nie sein kritisches Gebiet betritt und immer anderen den Vortritt lassen muß. Es ist nämlich völlig unbestimmt, wer (in Zeile 15) erfolgreich k auf seine eigene Prozeßnummer setzt. Im schlimmsten Fall sind es „immer die anderen“, so daß ein Prozeß verhungern kann, obwohl er nicht blockiert ist (das ist ein „livelock“ [KWON78]).

Um diesen Mangel zu beheben, d.h. jedem Prozeß zu garantieren, nach endlich vielen Versuchen sein kritisches Gebiet betreten zu können, hat Knuth ([Knu66]) die folgende Lösung vorgeschlagen: (zunächst in der Originalfassung angegeben)

```
   INTEGER ARRAY control [1:N]; INTEGER k;

Programm fuer Prozess_i:

   BEGIN   integer j;
   L0:   control[i]:=1;
   L1:   FOR j:=k step -1 UNTIL 1,
         N step -1 UNTIL 1 do
         BEGIN
           IF j=i THEN GOTO L2;
           IF control[j] <> 0 THEN GOTO L1
         END;
   L2:   control[i]:=2;
```

```
 FOR j:=N step -1 UNTIL 1 DO
     IF (j<>i) AND (control[j]=2) THEN GOTO L0;
L3:  k:=i;
 critical section;
 k:=IF i=1 THEN N else i-1;
L4:  control[i]:=0;
L5:  remainder of cycle in which stopping is allowed;
 GOTO L0
 END
```

Knuth verwendet statt der beiden booleschen Felder b und c (s. Dijkstra) ein integer-Feld *control*, dessen Elemente die Werte 0, 1 und 2 annehmen können und mit 0 initialisiert werden. Die beiden Datenstrukturen lassen sich aber leicht zueinander in Beziehung setzen, so daß die eine Konstruktion stets durch die andere ersetzt werden kann:

Dijkstra	Knuth	Bedeutung
`b[i]=true`	control[i]=0	Prozeß i gehört nicht zu den Bewerbern um den Eintritt in ihr kritisches Gebiet.
`(b[i]=false) && (c[i]=true)`	control[i]=1	Prozeß i ist noch nicht im engeren Kreis der Bewerber.
`(b[i]=false) && (c[i]=false)`	control[i]=2	Prozeß i steht unmittelbar vor dem Eintritt in sein kritisches Gebiet oder hat es bereits betreten.

Wir wollen auch jetzt wieder das vorgelegte Programm in ein goto-freies Pascalprogramm umsetzen und statt der booleschen Felder entsprechende Mengen verwenden, wobei: $i \in b$ entspricht `b[i] = true` und $i \in c$ entspricht `c[i] = true`.

```
(*  1 *) PROGRAM process_i (b, c, k);
(*  2 *) CONST i = ...;
(*  3 *)       N = ...;
(*  4 *) TYPE  T = 1..N;
(*  5 *) VAR   b, c : SET OF T;
(*  6 *)       (* Initialisierung : b:=[1..N], c:=[1..N]
(*  7 *)       k    : T;
(*  8 *)       ok   : boolean;
(*  9 *) BEGIN
(* 10 *) WHILE true DO BEGIN
(* 11 *)    b:=b-[i];
(* 12 *)    REPEAT
(* 13 *)       j:=k;   c:=c+[i];
(* 14 *)       WHILE j <> i DO
(* 15 *)          IF j IN b THEN
(* 16 *)             BEGIN
(* 17 *)             j:=j-1;
(* 18 *)             IF j = 0 THEN j:=N
(* 19 *)             END
(* 20 *)          ELSE   j:=k;
```



```
(* 21 *)       c:=c-[i];
(* 22 *)       ok:=(c+[i]) = [1..N];
(* 23 *)    UNTIL ok;
(* 24 *)    k:=i;
(* 25 *)    critical section;
(* 26 *)    IF i = 1 THEN k:=N ELSE k:=i-1
(* 27 *)    c:=c+[i];
(* 28 *)    b:=b+[i];
(* 29 *)    remainder of cycle in which stopping is allo
(* 30 *) END
(* 31 *) END.
```

Der gegenseitige Ausschluß ist gewährleistet, es ist also höchstens ein einziger Prozeß in seinem kritischen Gebiet.

Beweis: Ein Prozeß, der sein kritisches Gebiet betreten will, muß die repeat-Schleife (Zeilen 12 bis 23) erfolgreich durchlaufen, d.h. sein letzter Test `ok:=(c+[i])=[1..N]` muß den Wert *true* ergeben. Hier wird getestet, ob der Prozeß i der einzige Prozeß ist, der nicht mehr in der Menge c ist. Bevor er diesen Test ausführt, hat er sich selbst aus der Menge c entfernt, d.h. jeder Prozeß, der bis zu diesem Test vorstößt, ist nicht mehr in c. Deswegen kann höchstens ein Prozeß den Wert *true* erhalten, deswegen die repeat-Schleife verlassen und sein kritisches Gebiet betreten.

Man muß beachten, daß es möglich ist, daß mehrere Prozesse gleichzeitig bis `c:=c-[i]` in Zeile 21 kommen können.

Zu einem beliebigen Zeitpunkt sei $b = [1..N]$, $c = [1..N]$ und $k = N$. Als erster möge Prozeß 1 beginnen. Er wird die while-Schleife für `j`=N, $N-1, \ldots 2$, 1 durchlaufen und dann verlassen. Wenn er bei Zeile 21 ankommt, möge Prozeß 2 denselben Test durchlaufen. Da immer noch $k = N$ ist und $b = [2..N]$, durchläuft auch er die while-Schleife für `j`=N, $N-2, \ldots 3$, 2 und ist dann ebenfalls bei Zeile 21. Genauso geht es den Prozessen 3, 4, $\ldots N-1$, N (in dieser Reihenfolge). In diesem speziellen Beispiel ist es also allen Prozessen gelungen, bis zu Zeile 21 vorzustoßen.

Falls zwei oder mehr Prozesse auf den Eintritt in ihr kritisches Gebiet warten, wird einem davon nach endlicher Zeit der Zutritt gewährt.

Beweis: Wenn nur ein einziger konkurrierender Prozeß existiert, dann wird er die while-Schleife solange durchlaufen bis er $j = i$ findet (k kann dabei jeden beliebigen Wert haben), dann findet er, daß alle außer ihm in der Menge c sind und betritt sein kritisches Gebiet.

Knüpfen wir nun aber an die unangenehmere Situation des letzten Beispiels an. Im ungünstigsten Fall werden alle den Test `ok:=(c+[1])=[1..N]` in Zeile 22 erfolglos durchlaufen, d. h. alle Prozesse finden sich in der while-Schleife ab Zeile 10 wieder. Da k inzwischen nicht geändert werden konnte, werden alle Prozesse $i \neq k$ in `while i<>j do ... j:=k` auf der Stelle treten. Sie müssen alle den else-Zweig betreten, da $j = k$ und $j \notin b$ ist, weil ja Prozeß k einer der konkurrierenden Prozesse ist.

Nur dem Prozeß k, also mindestens einem Prozeß, wie behauptet, gelingt es nun, die while-Schleife zu verlassen und festzustellen, daß er als einziger nicht in der Menge c ist. Deswegen betritt er nun sein kritisches Gebiet.

Jedem auf Eintritt in sein kritisches Gebiet wartenden Prozeß gelingt dieser Eintritt nach endlich langer Zeit.

Beweis: Ein Prozeß muß auf höchstens $2^{N-1} - 1$ Besuche anderer Prozesse ihres kritischen Gebietes warten bis er selbst eintreten kann [ZIMA76].

5.3.1 Modifikationen

Um einem Prozeß eine kürzere Wartezeit als $2^{N-1} - 1$ Umläufe zu garantieren, wurden mehrere Modifikationen vorgenommen.

Von de Bruijn ([dB67]) stammt der Vorschlag, im Algorithmus von Knuth die Anweisungsfolge (die Zeilen 24 bis 26)

```
k:=i;
critical section;
IF i=1 THEN k:=N ELSE k:=i-1;
```

zu ersetzen durch

```
critical section;
IF (k IN b) OR (k=i) THEN
   IF k=1 THEN k:=N ELSE k:=k-1;
```

Damit sind weiterhin alle Anforderungen erfüllt, aber die maximale Wartezeit beträgt nur noch $N \cdot (N-1)/2$ Umläufe.

Eine weitere Modifikation stammt von Eisenberg und McGuire ([EMG72]). Im wesentlichen läuft ihr Vorschlag darauf hinaus, die Anweisung in Zeile 26

```
IF i=1 THEN k:=N ELSE k:=i-1;
```

zu ersetzen durch

```
j:=k;
REPEAT
   IF j=N THEN j:=1 ELSE j:=j+1
UNTIL NOT (j IN b);
k:=j;
```

Damit wird die maximale Wartezeit auf $N - 1$ Umläufe reduziert. (Der Vorschlag von Eisenberg und McGuire lautet im Original:

```
FOR j:=k STEP 1 UNTIL N,
   1 STEP 1 UNTIL k DO
    IF j<>k AND control[j]<>0 THEN
   BEGIN k:=j; GOTO L6 end;
L6: control[i]:=0;
```

Man überlege sich, daß die repeat-Schleife dasselbe leistet.)

5.3.2 Lösung nach Lamport

Ein viel einfacheres Programm, das die von Dijkstra gestellte Aufgabe löst, hat Lamport ([Lam76]) angegeben. Auffallend an dieser Lösung ist, daß alle Prozesse auf Daten anderer Prozesse nur lesend zugreifen und nur die eigenen Daten verändern. In den bisher vorgestellten Versionen griffen alle Prozesse auf eine gemeinsame Größe k sowohl lesend als auch schreibend zu. Wenn die Rechnereinheit, in der k gespeichert ist, ausfällt, dann bricht das ganze System zusammen. In der neuen Lösung kann man davon ausgehen, daß jeder der N Prozesse eine eigene Speichereinheit besitzt. Es gibt noch einen weiteren grundsätzlichen Unterschied zu den bisher angegebenen Algorithmen, die Prozesse betreten hier nämlich das kritische Gebiet in „first-come-first-served"-Ordnung. Dazu gibt jeder Prozeß sich selbst eine Zahl, die größer ist als die entsprechenden Zahlen der anderen Prozesse und die seine Priorität bestimmt. Sollten sich zwei Prozesse dieselbe Zahl zugeteilt haben – das ist möglich – dann soll der Prozeß mit der niedrigeren Prozeßnummer bevorzugt sein. Die Relation „kleiner als" ist für ein geordnetes Paar von ganzen Zahlen definiert durch:

$$(a, b) < (c, d), \text{ wenn } a < c \text{ oder wenn } a = c \text{ und } b < d.$$

Das (pascalähnliche) Programm für den Prozeß i lautet:

```
PROGRAM process_i(number, choosing);
CONST  i = ... ;   (* Prozessnummer *)
       N = ... ;   (* Anzahl der Prozesse *)
TYPE   T = 1..N;
VAR    j : T;
  number : ARRAY[ T ] OF integer;
         (* initialisiert mit 0 *)
  choosing : ARRAY[ T ] OF boolean;
         (* initialisiert mit false *)

BEGIN
  WHILE true DO BEGIN
    choosing[i]:=true;
    number[i]:=1 + maximum( number[1] .. number[N] );
    choosing[i]:=false;
    FOR j:=1 TO N DO BEGIN
      WHILE choosing[j] DO ;  (* wait *)
      WHILE (number[j]=0) AND
            ((number[j],j) < (number[i],i)) DO ;   (* wa
    END  (* for *);
    critical section ;
    number[i]:=0;
    noncritical section ;
  END  (* while *)
END  (* process_i *) .
```

Ein Prozeß i ist Mitbewerber um das kritische Gebiet, sobald er *choose[i]* auf *true* gesetzt hat, um dann mit der Suche des Maximums von *number* zu beginnen. Er hat sich selbst einen Prioritätswert zugeteilt, wenn er *choose[i]* wieder auf *false* setzt. Diese Operationen können gleichzeitig von mehreren Prozessen ausgeführt werden, so daß man nicht ausschließen kann, daß deren *number[j]* gleich sind. Wenn der Prozeß j nach Prozeß k diese Programmstelle durchläuft, dann ist offensichtlich `number[j] >= 1 + number[k]`. Die anschließende for-Schleife garantiert, daß nur ein einziger Prozeß das kritische Gebiet betritt, denn es kann nur einen einzigen Prozeß i geben, für den

```
(number[j],j) >= (number[i],i) AND 1 <= j <= N
```

gilt. Es ist übrigens denkbar, daß der Wert von *number[i]* zu groß wird und dann überläuft, nämlich genau dann, wenn es immer Bewerber um das kritische Gebiet gibt. Andernfalls kann die Zählung wieder bei Null beginnen. Aber selbst dann, wenn ununterbrochen, d.h. 24 Stunden am Tag, nach jeweils 1 msec *number[i]* um 1 erhöht wird, tritt bei `maxint` $= 2^31 - 1$ (wie bei Siemens) erst nach ca. 25 Tagen der Überlauf auf. In der Praxis wird dieser Fall nie eintreten.

5.3.3 Lösung nach Peterson

??? Eine weitere, wiederum einfachere Lösung stammt von G. L. Peterson ([Pet81]), für die es auch einen ausführlichen Beweis gibt [???]. Es bewerben sich wieder N Prozesse P_1 bis P_n um den Eintritt in ein kritisches Gebiet. Dazu greifen sie auf zwei globale Arrays q und *turn* zu, wobei wir annehmen, daß (nur) Lese- und Schreiboperationen auf dem Speicher unteilbar sind. Für die Arrayzugriffe besitzt jeder Prozeß die beiden lokalen Variablen j und k, die zum Indizieren verwendet werden:

```
PROGRAM process_i();

WHILE true DO BEGIN

   (* noncritical part of process_i *)

   FOR j:=1 TO N-1 DO BEGIN

      q[i]:=j;
      turn[j]:=i;

   L: FOR k:=1 TO N DO BEGIN
         IF k=i GOTO L;
         IF ((q[k]>=j) AND (turn[j]=i)) GOTO L;
      END;

      q[i]:=N;
      (* critical section of process_i *)
      q[i]:=0;
```



```
      END;

   END;
```

Um die Technik zu durchschauen, ist es am einfachsten, den Sonderfall $N = 2$ aus der allgemeinen Lösung zu gewinnen. Wie man sofort sieht, wird jeder Prozeß die globale Variable `turn[i]` auf den Wert der jeweils anderen Prozeßnummer setzten. Eine Indizierung ist also nicht mehr nötig.

6. Synchronisation mit Semaphoren [DIJK68]

Die in Kapitel 5 beschriebenen Verfahren sind viel zu schwerfällig und zu undurchsichtig, als daß sie in der Praxis (außer in Ausnahmesituationen) eingesetzt werden könnten. Außerdem würden dabei die an der Synchronisation beteiligten Prozesse während des Wartens auf ein Ereignis durch die ständigen Abfragen aktiv beschäftigt sein (busy waiting). Ökonomischer wäre es, wenn die Prozesse während des Wartens „schlafen“ gingen und beim Eintreten des Ereignisses, worauf sie warten, geweckt würden.

Ein elegantes und gut zu implementierendes Verfahren hat E. W. Dijkstra im Jahre 1968 vorgeschlagen ([Dij68]). Er führte einen neuen Datentyp „Semaphor“[1] ein. Dieser hat als Wertebereich die nicht negativen ganzen Zahlen. Die beiden neben der stets erforderlichen Initialisierung auf ihnen zulässigen Operationen sind $P(s)$ und $V(s)$, die wie folgt definiert sind und als unteilbare Elementaroperationen betrachtet werden:

$P(s)$: `s:=s-1`

$V(s)$: `s:=s+1`

Bei der P-Operation wird der Wert von s um 1 vermindert, sofern $s > 0$ ist. Falls $s = 0$ ist, kann die Subtraktion nicht ausgeführt werden, da sonst s den zulässigen Wertebereich verließe. Der Prozeß wird deswegen angehalten. (Wäre s eine Pascal-Variable, `s:0..maxint`, so träte ein Laufzeitfehler auf. Man beachte den Unterschied!) Er wird erst dann fortgesetzt, wenn von einem anderen Prozeß dieses Semaphor s um 1 (in einer $V(s)$-Operation) erhöht wird. Sollten mehrere Prozesse gleichzeitig auf s warten, so wird einer von ihnen (ein einziger) fortgesetzt. Es ist nicht festgelegt, welcher Prozeß das ist.

Das im letzten Kapitel behandelte Synchronisierungsproblem läßt sich jetzt sehr einfach lösen:

```
VAR mutex: semaphore;  (* mutex = MUTual EXclusion *)
BEGIN
   init(mutex:=1);  (*Initialisierung*)

   PARBEGIN
      process_1 : CYCLE
      BEGIN
         P(mutex);
         critical section 1;
         V(mutex);
```

[1] Semaphor [griechisch], Signaleinrichtung, die Nachrichten durch verschiedene Stellungen beweglicher Arme übermittelt.


```
         remainder of cycle 1
      END;

      process_2 : CYCLE
      BEGIN
         P(mutex);
         critical section 2;
         V(mutex);
         remainder of cycle 2
      END;

   PAREND
END.
```

Entsprechend ihrem Wertebereich unterscheidet man zwei Semaphortypen:

- `TYPE binary semaphore = 0..1; (* binäres Semaphor *)`
- `TYPE general semaphore = 0..u; (* allgemeines Semaphor *)`

wobei u beliebig groß werden kann. Binäre Semaphore dienen in erster Linie zum Schützen kritischer Gebiete (gegenseitiger Ausschluß). Man kann zeigen, daß man auf die allgemeinen Semaphore verzichten könnte, da man sie mit einigem Aufwand auf die binären Semaphore zurückführen kann. Wir wollen aber in der Anwendung keinen Unterschied zwischen diesen beiden Typen machen.

Bevor wir die P- und V-Operationen für die Synchronisierung anwenden, soll zunächst eine andere Form dieser Operationen angegeben werden, die der Implementierung gemäßer ist. Es gelten aber weiterhin $P(s)$ und $V(s)$ als unteilbare Operationen.

In der ersten Fassung wird noch die ursprüngliche Definition der Semaphorvariablen verwendet.

```
P(s) :: IF s = 0 THEN warte( s );
        s:=s-1
```

und

```
V(s) :: s:=s+1; (* Aus der Definition folgt, dass s:=s+1;
                   identisch ist mit s:=1; *)
        IF "jemand wartet auf s" THEN wecke (s);
```

Es soll jetzt angegeben werden, wie „warte“ und wie „wecke“ zu verstehen ist.

`warte(s)` soll bedeuten, daß derjenige Prozeß, der die P-Operation ausführen will, angehalten wird. In seiner Zustandsbeschreibung wird notiert, daß er auf eine V-Operation auf das Semaphor s wartet. Bei der Ausführung

von $V(s)$ sieht das Betriebssystem in `wecke(s)` nach, ob es einen Prozeß gibt, der auf die Erhöhung dieses Semaphors s wartet. Wenn das der Fall ist, dann wird dieser geweckt, d.h. er wird aus seinem Wartezustand erlöst. Er kann sofort weiterrechnen. Wenn mehrere Prozesse auf dasselbe Semaphor warten, dann darf nur einer von ihnen geweckt werden. Dijkstra ([Dij68]) schlägt vor, daß aus der Menge der wartenden Prozesse irgendeiner zufallssmäßig ausgewählt wird. Er läßt aber auch gelten ([Dij71]), daß die Auswahl z.B. nach dem FIFO-Prinzip durchgeführt wird. In beiden Fällen funktioniert die Synchronisierung, dennoch gibt es einen wichtigen Unterschied. Während nämlich das Wecken bei zufallsmäßiger Auswahl beliebig lange verzögert werden kann, garantiert die FIFO-Auswahl das Wecken nach einer endlich langen und fairen Wartezeit [STAR82] (Wenn man die verschiedenen Versionen und Methoden der Synchronisierung auch im Detail miteinander vergleichen will, dann genügt eine informelle Beschreibung, wie sie hier gegeben wird, nicht mehr, dann ist eine strenge Formalisierung erforderlich.)

Für die Implementierung ist es geschickter, sich von der ursprünglichen Definition des Semaphor-Datentyps frei zu machen und auch negative Werte zuzulassen. Man kommt dann zu folgender Form, die in der Wirkung mit der vorher genannten gleichwertig ist:

```
P(s) :: s:=s - 1;
        IF s<0 THEN warte (s);
```

und

```
V(s) :: s:=s + 1;
        IF s<=0 THEN wecke (s);
```

In dieser Fassung ist von Vorteil, daß das Betriebssystem an der Größe von s erkennen kann, ob und wieviele Prozesse auf $V(s)$ warten. (Hinweis: Die Beschreibung von P- und V-Operationen ist in Lehrbüchern oft falsch dargestellt.

6.1 Das Erzeuger-Verbraucher-Problem ([Dij68])

Bei dem Erzeuger-Verbraucher-Problem handelt es sich um die Kommunikation, d.h. den Nachrichtenaustausch zwischen Erzeugern, die Nachrichten produzieren, und Verbrauchern, die diese Nachrichten verarbeiten. Unter der Annahme der Gleichberechtigung von Erzeugern und Verbrauchern reicht zur Übermittlung der Botschaften ein einziger globaler Pufferspeicher aus. Bei der Lösung muß sichergestellt sein, daß

1. Zugriffe auf den Puffer sich nicht gegenseitig stören.
2. Nichts in den Puffer gelegt wird, wenn er voll ist.
3. Dem Puffer nichts entnommen wird, wenn er leer ist.
4. Über die Arbeitsgeschwindigkeit von Erzeugern und Verbrauchern keine Annahmen gemacht werden müssen (Zeitunabhängigkeit).

Diese Forderungen sind offensichtlich in der jetzt angegebenen Lösung erfüllt:

Erzeuger : Lege Nachricht in den Puffer, falls er leer ist; melde, daß eine Nachricht vorhanden ist.

Verbraucher : Hole Nachricht aus Puffer, falls eine Nachricht da ist; melde, daß der Puffer jetzt leer ist.

Diese verbale Beschreibung führt zu folgendem Programm:

```
VAR message, leer : semaphore;

init(message:=0);
init(leer:=1);

erzeuger:
   erzeuge Nachricht;
   P(leer);
   lege Nachricht in Puffer;
   V(message);

verbraucher:
   P(message);
   hole Nachricht aus Puffer;
   V(leer);
   verarbeite Nachricht;
```

Durch diese Lösung wird eine sequentielle Reihenfolge erzwungen, so daß die Synchronisierung zwangsläufig ist. Hier wird außerdem angenommen, daß das Erzeugen und das Verarbeiten von Nachrichten außerhalb des gemeinsamen Puffers möglich ist. Das ist aber oft, z. Bsp. bei der Versorgung von E/A-Geräten nicht der Fall. Mit diesen Einschränkungen kommt man dann zu folgender Lösung:

```
VAR message, leer : semaphore;
init (message:=0);
init (leer:=1);

erzeuger:
   P(leer);
   erzeuge Nachricht;
   lege sie in den Puffer;
   V(message);

verbraucher:
   P(message);
   hole und verarbeite Nachricht;
   V(leer);
```

Hier wurden also auch das Erzeugen und Verarbeiten der Nachrichten in den geschützten Bereich gelegt. Im übrigen wurde wieder eine sequenti-

elle Bearbeitungsfolge erzwungen. Nehmen wir einmal an, der Verbraucher sei ein Prozeß, der eine Nachricht aus dem Puffer holt und sie verarbeitet, indem er sie auf einen Drucker gibt. Der Erzeuger holt z.B. Information aus einer Datei, bereitet sie auf zum Drucken, und legt sie in den Druckpuffer. Dies kann aber in der hier angegebenen Lösung erst geschehen, wenn zuvor der Druckpuffer geleert wurde. Die Folge ist, daß der Drucker „stottert", denn er kann ja erst dann weiterarbeiten, wenn vom Erzeuger wieder Information aufbereitet worden ist. Um das Stottern zu vermeiden, geht man zur Doppelpuffertechnik über. Man verwendet jetzt nicht mehr nur einen einzigen Puffer, sondern zwei Puffer, die in abwechselnder Reihenfolge gefüllt und geleert werden. Während also Puffer 1 geleert wird, wird der andere, Puffer 2, gefüllt. Nehmen wir an, Puffer 2 ist eher gefüllt als Puffer 1 geleert. Dann muß der Erzeuger anhalten. Dagegen kann der Verbraucher nach Leeren von Puffer 1 ohne Unterbrechung weiterarbeiten, da er dann ja einen gefüllten Puffer 2 vorfindet. Außerdem muß er dem Erzeuger mitteilen, daß ihm jetzt Puffer 1 zum Füllen zur Verfügung steht. Das Füllen des einen und das Leeren des anderen Puffers kann gleichzeitig (parallel) erfolgen.

Wir wollen sofort eine Verallgemeinerung vornehmen und auf k Puffer übergehen bzw. zu einem Puffer, der in k Einheiten unterteilt und als Kreispuffer organisiert ist.

```
VAR leer, voll : semaphore;
puffer : ARRAY[0..k-1] OF information;
ex : integer; (*Erzeugerindex*)
vx : integer; (*Verbraucherindex*)

ex:=0;
vx:=0;
init(leer:=k);
init(voll:=0);

erzeuger:
   erzeuge Nachricht;
   p(leer);
   puffer[ex]:=erzeugte Nachricht;
   ex:=succ(ex); (*modulo k*)
   v(voll);

verbraucher:
   p(voll);
   hole Nachricht aus puffer[vx];
   vx:=succ(vx); (*modulo k*)
   v(leer);
   verarbeite Nachricht;
```

Mit $k = 2$ ergibt sich die Doppelpuffertechnik. Ob die angegebene Lösung wirklich die gestellten Forderungen erfüllt, muß noch untersucht werden. Der Beweis wird in einem späteren Kapitel geführt.

6.2 Allgemeine/binäre Semaphore

Wie bereits erwähnt wurde, kann man allgemeine Semaphore auf binäre Semaphore zurückführen. Um das zu zeigen, wollen wir an das Erzeuger-Verbraucher-Problem in einer vereinfachten Form anknüpfen. Wegen der Symmetrie der Eigenschaften von Erzeuger und Verbraucher genügt es, das Lesen der Nachricht aus dem Puffer zu behandeln, d.h. der Puffer ist in diesem Beispiel beliebig groß.

Die Aufgabe besteht nun darin, die Lösung mit allgemeinen Semaphoren:

```
Erzeuger                          Verbraucher
---------------                   ---------------
{lege Nachricht in Puffer}        P(message)
V(message)                        {hole Nachricht aus Puffe
```

in eine Lösung mit binären Semaphoren umzuwandeln.

Zusätzlich zu einem Semaphor für die Synchronisierung braucht man jetzt offenbar noch einen Zähler für die Anzahl der Nachrichten im Puffer. Im ersten Ansatz:

```
Erzeuger                          Verbraucher
--------                          -----------
{lege Nachricht in Puffer}        IF count=0 THEN P(s);
count:=count+1;                   {hole Nachricht aus Puffer
IF count=1 THEN V(s);
```

wurde offensichtlich vergessen, den Zugriff auf die gemeinsame Variable *count* in ein kritisches Gebiet zu legen, damit der Zugriff darauf exklusiv erfolgen kann. Man braucht also noch ein Semaphor für den gegenseitigen Ausschluß und kommt zum nächsten Ansatz:

```
Erzeuger                          Verbraucher
--------                          -----------
{lege Nachricht in Puffer}        P(mutex)
P(mutex)                          IF count=0 THEN P(s);
count:=count+1;                   count:=count-1
IF count=1 THEN V(s)              {hole Nachricht aus Puffe
V(mutex)                          V(mutex) 1
```

Diese Lösung führt zwangsläufig zum Deadlock. Sobald nämlich der Verbraucher festgestellt hat, daß `count=0` ist, wird er in $P(s)$ anhalten ohne dem Erzeuger Gelegenheit zu geben, den Zähler zu erhöhen und einen wartenden Verbraucher zu wecken, da niemand mehr an *P(mutex)* vorbeikommt. Der Verbraucher muß also, bevor er in den Wartezustand übergeht, das mit dem Semaphor *mutex* geschützte kritische Gebiet freigeben:

```
Erzeuger                          Verbraucher
--------                          -----------
{lege Nachricht in Puffer}        P(mutex)
P(mutex);                         IF count=0 THEN
count:=count+1;                   BEGIN
```

```
IF count=1 THEN V(s);
V(mutex);
```

```
   V(mutex);
   P(s);
   P(mutex)
END;
count:=count-1;
{hole Nachricht aus Puffe
V(mutex);
```

Daß aber auch diese Lösung falsch ist, ist nicht auf den ersten Blick zu erkennen. Nehmen wir einmal folgende Aktionsfolge an:

1. Start mit `mutex=1`, `s=0`, `count=0`.
2. Erzeuger legt Nachricht in den Puffer. Er stellt fest, daß dies die erste Nachricht war und führt `V(s)` aus, d.h. `s=1`, `count=1`.
3. Der Verbraucher findet `count<>0` und holt sofort die Nachricht aus dem Puffer: `count=0`, `s=1`.
4. Der Verbraucher will die nächste Nachricht holen. Er stellt `count=0` fest und durchläuft:
   ```
   V(mutex);
   P(s); (* er wartet nicht, da ja s=1 war!  *)
   P(mutex);
   count:=-1;
   ```
   und will dann eine Nachricht holen, die es gar nicht gibt.

Offenbar ist die $V(s)$-Operation vom Erzeuger grundlos gegeben worden, denn mit $V(s)$ soll jetzt nicht mehr wie zuvor in der Lösung mit allgemeinen Semaphoren signalisiert werden, daß eine Nachricht vorliegt, sondern es soll ein auf eine Nachricht wartender Verbraucher geweckt werden. $V(s)$ darf also nur ausgeführt werden, wenn ein Verbraucher wartet, sonst nicht. Diese Forderung ist leicht erfüllt, wenn die Abfrage auf *count* geändert wird:

```
Erzeuger                          Verbraucher
--------                          -----------
{lege Nachricht in Puffer}        P(mutex)
P(mutex);                         count:=count-1;
count:=count+1;                   IF count < 0 THEN
IF count<=0 THEN V(s);            BEGIN
V(mutex);                            V(mutex);
                                     P(s);
                                     P(mutex)
                                  END;
                                  {hole Nachricht aus Puffe
                                  V(mutex);
```

Hinweis: Man vergleiche diese Lösung mit der Definition von P- und V-Operationen auf allgemeine Semaphore!

Damit ist eine Lösung angegeben, die mit binären Semaphoren auskommt. Wenn man die Definition eines Semaphors (sein Wert darf nur ≥ 0 sein)

mit dem Wertebereich von *mutex* vergleicht, so stellt man fest, daß *mutex* auch bei beliebig vielen Verbrauchern und Erzeugern immer nur die Werte 0 und 1 annehmen kann. Merkwürdiger ist das Verhalten des Semaphors s; es hat nämlich immer den Wert Null! Die Operation $P(s)$ wird nur ausgeführt, um in den Wartezustand überzugehen und $V(s)$ wird nur ausgeführt, um einen wartenden Prozeß zu wecken.

Schließlich soll noch eine andere Lösung angegeben werden, die noch eine zusätzliche (lokale) boolesche Größe *leer* verwendet, die mit `true` initialisiert ist:

```
Erzeuger                          Verbraucher
--------                          -----------
{lege Nachricht in Puffer}        IF leer THEN P(s);
P(mutex);                         P(mutex)
count:=count+1;                   count:=count-1;
IF count=1 THEN V(s)              leer:=count=0;
V(mutex);                         {hole Nachricht aus Puf
                                  V(mutex);
```

Im Gegensatz zur zuletzt angegebenen Lösung werden hier $P(s)$- und $V(s)$-Operationen öfter als notwendig ausgeführt, nämlich immer dann, wenn zwischenzeitlich `count=0` geworden war. Zum Zeitpunkt der Abfrage auf *leer* können aber schon wieder Nachrichten im Puffer liegen.

6.3 Das Leser-Schreiber-Problem

Eines der Standard-Beispiele ist das von Courtois, Heymans und Parnas ([CHP71])behandelte und später von Brinch Hansen [BRIN71] aufgegriffene Leser-Schreiber-Problem. Davon gibt es von der Aufgabenstellung her zwei Modifikationen, die hier in Anlehnung an Courtois, Heymans und Parnas wiedergegeben werden sollen.

6.3.1 Problem 1

Wir haben es hier mit zwei Klassen von Prozessen zu tun, nämlich den Lesern und den Schreibern. Prozesse beider Klassen greifen auf ein gemeinsames Betriebsmittel zu, z.B. eine Datei. Schreiber benötigen exklusiven Zugriff, Leser dagegen nicht. Wenn ein Schreiber auf diese gemeinsamen Betriebsmittel zugreift, dann ist jeder weitere Zugriff für alle Leser und auch für alle weiteren Schreiber verboten. Sollte sich während des Zugriffs von Lesern ein Schreiber bewerben, so muß er warten bis kein Leser mehr aktiv ist.

Die gestellte Aufgabe wird von der folgenden Konstruktion gelöst:

```
VAR readcount : integer;
mutex, w : semaphore;

readcount:=0;
init(mutex:=w:=1); (*Initialisierung der Semaphore*)
```

```
reader:
   P(mutex);
   readcount:=readcount + 1;
   IF readcount = 1 THEN P(w);
   V(mutex);
      lesender Zugriff
   P(mutex);
   readcount:=readcount-1;
   IF readcount = 0 THEN V(w);
   V(mutex);

writer:
   P(w);
      schreibender Zugriff
   V(w);
```

Das Semaphor *mutex* braucht man, um den Zugriff auf die Variable *readcount* zu koordinieren. Alle Manipulationen an *readcount* liegen damit in einem geschützten kritischen Gebiet.

Man braucht diese integer-Größe, damit der erste Leser und nur dieser durch die $P(w)$-Operation entweder den gesamtem Lesestrom anhält, falls gerade geschrieben wird, oder aber schreibende Zugriffe unmöglich macht. In ähnlicher Weise gibt der letzte Leser nach getaner Arbeit mit $V(w)$ Schreibzugriffe frei.

Trotz richtiger Synchronisierung ist diese Lösung unbefriedigend. Angenommen, ein Schreiber arbeitet. Weitere Schreiber und der erste Leser werden dann in ihren jeweiligen $P(w)$-Operationen stehenbleiben und warten. Wenn dann der Schreiber seine Arbeit mit $V(w)$ beendet, dann kann man auf Grund der Definition der P/V-Operationen nicht sagen, wer geweckt wird. Es gibt keine Prioritäten für Leser und Schreiber. Schlimmer ist, daß die Möglichkeit besteht, daß Schreiber „verhungern“, nämlich dann, wenn während ein Leser tätig ist, mindestens ein weiterer Leser dazukommt. Da in diesem Falle also immer mindestens ein Leser arbeitet, gibt es keinen „letzten“ Leser, der mit einer $V(w)$-Operation die Möglichkeit zum Schreiben erlauben würde. Die gestellte Aufgabe soll deswegen modifiziert werden.

6.3.2 Problem 2

Es wird jetzt gefordert, daß Schreiber bevorzugt behandelt werden. Sobald ein Schreiber auf die Aufnahme seiner Tätigkeit warten muß, dürfen keine neuen Leser mehr angenommen werden. Der Lesestrom wird angehalten. Die Anzahl der Leser kann jetzt nur noch abnehmen. Es wird also einen letzten Leser geben, der sich nach Beendigung seiner Arbeit dadurch verabschiedet, daß er das Schreiben erlaubt.

Von Courtois, Heymans und Parnas wurde folgende Lösung angegeben:


```
VAR readcount, writecount : integer;
mutex1, mutex2, mutex3, w, r: semaphore;

readcount:=writecount:=0;
init (mutex1:=mutex2:=mutex3:=w:=r:=1);

reader:
   P(mutex3);
   P(r);
   P(mutex1);
   readcount:=readcount + 1;
   IF readcount = 1 THEN P(w);
   V(mutex1);
   V(r);
   V(mutex3);

   Lesender Zugriff

   P(mutex1);
   readcount:=readcount - 1;
   IF readcount = 0 THEN V(w);
   V(mutex1);

writer:
   P(mutex2);
   writecount:=writecount + 1;
   IF writecount = 1 THEN P(r);
   V(mutex2);
   P(w);

   schreibender Zugriff

   V(w);
   P(mutex2);
   writecount:=writecount - 1;
   IF writecount = 0 THEN V(r);
   V(mutex2);
```

Die Semaphore *mutex1* und *mutex2* erlauben exklusiven Zugriff auf die beiden integer-Größen *readcount* bzw. *writecount*. Ohne *mutex3* könnte man nicht die absolute Priorität von Schreibern gegenüber Lesern garantieren. Denn dann wäre es möglich, daß auf r ein Schreiber und mehrere Leser warten. Es wäre also nicht klar, wer mit $V(r)$ geweckt wird. Wenn ein Leser das Gebiet zwischen $P(r)$ und $V(r)$ betreten hat, dann bleiben alle neuen Leser in P(*mutex3*) stehen. Die einzig mögliche $P(r)$-Operation kann nur von einem Schreiber ausgeführt werden, der dann allerdings warten muß. Das hat zur Folge, daß mit einem von einem Leser gegebenen $V(r)$ nur ein wartender Schreiber (falls vorhanden) geweckt werden kann und niemals ein Leser.

Bei genauerem Hinsehen stellt man jedoch fest ([Age77]), daß die gestellten Forderungen doch nicht immer erfüllt sein müssen. Eine Schwierigkeit entsteht dadurch, daß nicht genau gesagt ist, wodurch sich ein Schreiber bemerkbar machen soll. Es ist naheliegend anzunehmen, daß dies durch die Anweisung `writecount:=writecount + 1;` geschieht. Da aber beide Prozesse zeitlich unabhängig voneinander arbeiten, kann man keine Aussage machen, wieviel Zeit zwischen der Ausführung zweier Anweisungen vergeht. Wenn sich der erste Schreiber bis zur Ausführung von $P(r)$ Zeit läßt, dann können im Gegensatz zur Aufgabenstellung während dieser Zeit beliebig viele Leser das System betreten.

6.4 Das 5-Philosophen-Problem

Ein interessantes und lehrreiches Beispiel für die Synchronisation mit Semaphoren und für die daraus resultierenden zum Teil recht unangenehmen Konsequenzen stammt von Dijkstra ([Dij71]). Darin wird der Lebensrhythmus von 5 Philosophen betrachtet, die nacheinander denken und essen, denken, essen, denken, essen, usw.. Beide Tätigkeiten schließen sich gegenseitig aus. Das Leben des Philosophen i läßt sich nun wie folgt beschreiben:

```
philosoph_i:
CYCLE
   BEGIN
      denken;
      essen;
   END;
```

Die 5 Philosophen haben weniger Schwierigkeiten mit dem Denken als vielmehr mit dem Essen. Sie essen an einem Kreistisch, auf dem 5 Teller stehen. Zwischen jedem Teller liegt eine Gabel. Auf dem Tisch steht eine Schüssel mit Spaghetti. Jeder Philosoph benötigt entweder wegen der besonderen Art der Spaghetti oder aus eigener Ungeschicklichkeit zwei Gabeln zum Essen. Genauer gesagt, er braucht sowohl die Gabel links von seinem Teller (`Gabel[i]`) als auch die rechte Gabel (`Gabel[(i+1) mod 5`). Daraus folgt, daß höchstens zwei Philosophen gleichzeitig essen können und daß diese beiden nicht nebeneinander sitzen können. Da die Gabeln miteinander geteilt werden müssen, stellen sie ein exklusives Betriebsmittel dar, so daß der Zugriff darauf mit P- und V-Operationen synchronisiert werden muß. Die Philosophen und die Gabeln seien von 0 bis 4 indiziert. Naheliegend ist zunächst folgende Lösung:

```
philosoph_i:
CYCLE
   BEGIN
      denken;
      P(gabel[i]);            (*linke Gabel*)
      P(gabel[(i+1) mod 5]); (*rechte Gabel*)
      essen;
   END;
```


Diese Lösung kann offensichtlich nicht funktionieren. Wenn sich z.B. die 5 Philosophen gleichzeitig an den Tisch setzen und jeder seine linke Gabel (`gabel[i]`) nimmt, dann sind alle Gabeln vergeben. Es gibt keine freie rechte Gabel mehr. Die Philosophen können nicht essen, sie warten ewig und verhungern (Deadlock).

Es gibt eine einfache Methode, den Zugriff auf beide Gabeln zu ermöglichen. Man verlegt nämlich den Zugriff in ein geschütztes kritisches Gebiet

```
var mutex : binary semaphore;

philosoph_i:
CYCLE
  BEGIN
    denken;
    P(mutex); (*gegenseitiger Ausschlu"s*)
      P(gabel[i]);           (*linke Gabel*)
      P(gabel[(i+1) mod 5]); (*rechte Gabel*)
        essen;
      V(gabel[i]);
      V(gabel[(i+1) mod 5]);
    V(mutex);
  END;
```

Damit hat man allerdings auch erreicht, daß immer nur ein einziger Philosoph essen kann. Wenn man sich diese Lösung genauer ansieht, dann kann man sie auf

```
philosoph_i:
CYCLE
   BEGIN
     Denken;
     P(mutex);
       essen;
     V(mutex);
   END;
```

vereinfachen. Die Lösung jedoch ist zu restriktiv. Man kommt zu etwas besseren Verhältnissen, wenn man nur den Zugriff, nicht aber mehr das Essen in das kritische Gebiet legt:

```
philosoph_i:
CYCLE
   BEGIN
      denken;
      P(mutex);
        P(gabel[i]);
        P(gabel[(i+1) mod 5]);
      V(mutex);
      essen;
      V(gabel[i]);
      V(gabel[(i+1) mod 5]);
```

```
END;
```

Diese Lösung ist dennoch nicht optimal, denn es kann vorkommen, daß ein Philosoph warten muß, obwohl seine beiden Gabeln frei sind. In eine solche Situation kann z.B. Philosoph 2 geraten, wenn Philosoph 0 ißt (er belegt `gabel[0]` und `gabel[1]`) und Philosoph 1 essen will. Dieser versperrt das kritische Gebiet, weil er `gabel[1]` nicht erhält. Philosoph 2 könnte `gabel[2]` und `gabel[3]` benutzen, da sie frei sind, aber ihm ist der Eintritt in das kritische Gebiet versperrt.

Ähnliche Bemerkungen lassen sich zu folgender Lösung machen, die die Gabeln hierarchisch anordnet und verlangt, daß jeder Philosoph zuerst die Gabel mit der kleineren Nummer, und dann die Gabel mit der größeren Nummer ergreift. Diese Vorschrift führt zu einer deadlockfreien Lösung (s. Kapitel Deadlock). Das Programm für den Philosophen i lautet:

```
philosoph_i:
CYCLE
   BEGIN
      denken;
      P(gabel[min (i, (i+1) mod 5)]);
      P(gabel[max (i, (i+1) mod 5)]);
      essen;
      V(gabel[i]);
      V(gabel[(i+1) mod 5]);
   END;
```

Die Philosophen 1 bis 4 werden also zuerst die linke und dann die rechte Gabel nehmen, während Philosoph 5 erst die rechte und dann die linke Gabel nimmt.

Auch hier werden die Gabeln nicht optimal genutzt. Man muß also nach einer anderen Lösung suchen, die es einem Philosophen erlaubt, dann zu essen, wenn seine beiden Gabeln frei sind. Er muß dazu den Zustand der Gabeln abfragen können. Dies führt uns zu einer Lösung, die Zustandsvariablen verwendet.

Diese Zustandsvariablen sollen den Zustand eines Philosophen beschreiben. Dabei wird berücksichtigt, daß er nicht nur denkt oder ißt, sondern auch „hungrig“ ist, wodurch der Übergang des Zustandes „denken“ in den Zustand „essen“ eingeleitet wird:

`z[i]=0` : Philosoph i denkt.

`z[i]=1` : Philosoph i ist hungrig.

`z[i]=2` : Philosoph i ißt.

Alle $z[i]$ sollen mit 0 initialisiert sein. Der Übergang von $z[i] = 2$ nach $z[i] = 0$ ist offensichtlich ungefährlich, denn dabei gibt der Philosoph i seine Gabeln frei, eine Blockade kann nicht enstehen. Gefährlich ist dagegen der Übergang von $z[i] = 0$ nach $z[i] = 2$, weil dabei neue Betriebsmittel (Gabeln) verlangt werden und Philosoph i möglicherweise warten muß. Wenn alle Philosophen gleichzeitig essen wollen, kann es sein, daß

niemand essen kann und jeder auf die Nachbarn wartet. Der Übergang von $z[i] = 0$ nach $z[i] = 2$ muß also kontrolliert werden. Da nebeneinander sitzende Philosophen nicht gleichzeitig essen können, gilt folgendes Verbot:

```
(z[i]=2) AND (z[(i+1) MOD 5])=2
```

Jeder Philosoph wird die Zustände 0, 1, 2, 0, 1, ... zyklisch durchlaufen. Um vom Zustand 1 (hungrig) zum Zustand 2 (essen) zu wechseln, müssen folgende Bedingungen erfüllt sein:

1. `z[i]=1`: Philosoph i muß hungrig sein.
2. `z[(i+1) MOD 5] <> 2]`: sonst kann das o. a. Verbot verletzt werden.
3. `z[(i-1) MOD 5] <> 2]`: ebenso.

Es muß also ein i geben, für das gilt:

```
(z[(i-1) MOD 5] <> 2) AND (z[i] = 1) AND
   (z[(i+1) MOD 5] <> 2)
```

Damit nun der Philosoph i essen kann, muß er den Zustand seiner beiden Nachbarn prüfen. Vom Ergebnis der Prüfung hängt es ab, ob er essen darf oder ob er darauf warten muß, daß ihm beide benachbarten Gabeln zur Verfügung stehen. Falls er warten muß, können ihn nur diejenigen Philosophen aus dem Wartezustand befreien, die ihre Gabeln zurücklegen. Das kann dadurch geschehen, daß der Philosoph i beim Wechsel vom Zustand 2 in den Zustand 0 seine beiden Nachbarn testen läßt, ob sie essen wollen bzw. dürfen. Wir kommen damit zu folgendem Programm:

```
VAR mutex : semaphore;
    z : ARRAY [0..4] 0..2;
    i : integer;
    prisem : ARRAY [0..4] OF semaphore;

PROCEDURE test (k : integer);
BEGIN
   IF (z[(k-1) MOD 5] <> 2) AND (z[k] = 1) AND
      (z[(k + 1) MOD 5] <> 2)
   THEN BEGIN
      z[k]:=2; v(prisem[k])
   END
END  (* test *);

BEGIN
   init(mutex:=1);
   FOR i:=0 TO 4 DO z[i]:=0;
   FOR i:=0 TO 4 DO init(prisem[i]:=0);

(*Programm f"ur den Philosophen i*)
CYCLE
```

```
  BEGIN
    denken;
    P(mutex);
    z[i]:=1;       (*hungrig werden*)
    test(i);       (*darf man essen?*)
    V(mutex);
    P(prisem[i]); (*das Testergebnis entscheidet*)
    essen;
    P(mutex);
    z[i]:=0;        (*essen beendet*)
    test((i+1) MOD 5); (*will der rechte Nachbar essen?*)
    test((i-1) MOD 5); (*will der linke Nachbar essen?*)
    V(mutex)
  END
END.
```

Bemerkenswert an diesem Programm ist die Synchronisierung über „private Semaphore“ (prisem).

Die angegebene Lösung ist (ohne Beweis) richtig synchronisiert und frei von Verklemmungen. Es kann nicht vorkommen, daß jeder auf jeden wartet. Dennoch ist die Lösung nicht zufriedenstellend, denn es ist nicht garantiert, daß jeder (nach einer endlich langen Wartezeit) essen kann. Es ist möglich, daß durch eine unvorhersehbare Reihenfolge der Verplanung oder durch bewußte Herbeiführung einer solchen Reihenfolge einer der Philosophen, obwohl er hungrig ist, niemals essen kann („death by conspiracy“ ([Dij68]) oder „livelock“ [KWON78). Wenn z.Bsp. die Philosophen 2, 3 und 4 in dieser Reihenfolge hungrig werden und die Philosophen 0 und 1 nicht essen, dann können die Philosophen 2 und 4 essen, 3 muß warten. Er muß solange warten, bis weder Philosoph 2 noch 4 ißt. Wenn die beiden sich beim Essen abwechseln, dann muß Philosoph 3 verhungern.

6.5 Implementierung von P- und V-Operationen

In diesem Abschnitt soll gezeigt werden, wie Semaphore und die auf ihnen definierten Operationen implementiert und in ein Betriebssystem eingebettet werden können. Einige Annahmen über die Umgebung im Betriebssystem sind erforderlich, um die Vorgänge um die Begriffe „wecken“ und „warten“ zu verdeutlichen. Allerdings können auch hier einige Betriebssystemoperationen nur angedeutet werden, wie etwa das „Rechnendsetzen“, bei dem einem Prozeß die CPU entzogen und einem anderen Prozeß die CPU zugeteilt wird. Dieser Wechsel ist nicht ganz einfach zu implementieren, da ja für diesen Wechsel das Betriebbssystem selbst die CPU benötigt.

Für die Programmierung wollen wir folgende Annahmen machen:

- Der Rechner verfügt nur über eine einzige CPU, sodaß zwar eine Synchronisierung zwischen den Prozessen, nicht aber auch noch zwischen mehreren CPUs erforderlich ist.

- Alle Prozesse (Systemprozesse und Benutzerprozesse) sind eindeutig durch ihre Prozessnummer identifizierbar.
- in der Variablen *ego* wird stets notiert, welcher Prozeß gerade die CPU besitzt. Jeder Prozeß findet also seine eigene Prozeßnummer in der globalen Systemgröße *ego*. Bei der Regievergabe besetzt das Betriessystem diese Variabale mit der Prozeßnummer des neuen Prozesses.
- Alle Prozeßnummern sind der Wichtigkeit der Prozesse entsprechend (also nach Prioritäten) in einer Prioritätentabelle geordnet. Jede Priorität kann nur einmal vergeben werden. Sie ist dann nicht vergeben, wenn sie mit der Prozeßnummer Null besetzt ist. Der wichtigste Prozeß hat die Priorität *maxprio*, der unwichtigste die Priorität 1.
- Es gibt immer einen „unwichtigsten“ Prozeß. Das ist z.B. ein Systemprozeß, der nichts weiter tut als die CPU zu beschäftigen. Er besteht in der einfachsten Form aus der Anweisung `WHILE true DO;`
- Die beiden Prozeduren $P(s)$ und $V(s)$ werden (auf Maschinenebene) nicht mit dem Befehl `BAL` (branch and link) sondern mit dem Befehl `SVC` (supervisor call) aufgerufen. Beim Aufruf mit `SVC` werden gleichzeitig Interrupts gesperrt, so daß auch $P(s)$ und $V(s)$ als unteilbare Operation ausgeführt werden. Außerdem werden dabei die Privilegien geändert, die Prozesse gehen in den Systemmodus über, in dem Operationen erlaubt sind, die dem Benutzer sonst verschlossen sind.

Ausschnitt aus dem Vereinbarungsteil:

```
TYPE pzust =     (* Proze"szust"ande      *)
    (nvorh,      (* nicht vorhanden       *)
     rechnend,   (* im Besitz der CPU     *)
     bereit,     (* rechenwillig          *)
     wartend,    (* wartend auf V(s)      *)
     blockiert,  (* z.Bsp. vom Operateur  *)
     druckend,
     ...     );  (* andere Zust"ande      *)

     sname = integer;  (* Semaphorname *)

     pbschr =          (* Proze"sbeschreibung *)
       RECORD
         pnr : integer;          (* Proze"snummer     *)
         zust: SET OF pzust;     (* Zustand           *)
         sema: sname;            (* Semaphorname      *)
         ...                     (* weitere Kenndaten *)
       END;
```

```
VAR  prio : ARRAY[1..maxprio] OF 0..maxproz;
       (* Tabelle der Priorit"aten *) 2
     proz : ARRAY[0..maxproz] OF pbschr;

     s : ARRAY[1..maxs+1] OF  (* Tabelle der Semaphore *)
     RECORD
       sn : sname;     (* Semaphorname    *)
       sg : integer    (* Semaphorgr"o"se *)
     END;

     ego : 1..maxproz;      (* rechnender Proze"s *)
```

Beim Aufruf von P und V wird der Semaphorname als Parameter mitgegeben. Wir nehmen an, daß der Name als ganze Zahl codiert ist. Nimmt man stattdessen einen String, so ändert sich nur das Suchprogramm, das das Semaphor in der Tabelle der Semaphore sucht. Eine Funktion `seek(x)` erfüllt diese Aufgabe.

```
FUNKTION seek (x: sname): integer;
VAR i: integer;
BEGIN
   i:=0;
   s[maxs+1].sn:=x; (* Bremse beim Suchen *)
   REPEAT
      i:=i+1;
   UNTIL s[i].sn= x;
   seek:=i
END;
```

(Man wird in einem größeren Betriebssystem natürlich effizientere Suchmethoden wählen.)

Die Prozeduren $P(s)$ und $V(s)$ müssen u.U. auf die Prozeßbeschreibung zugreifen. Wenn ein Prozeß in $P(s)$ warten muß, dann soll in die Prozeßbeschreibung dieses Prozesses der Zustand „wartend“ eingetragen werden und auch der Name des Semaphors, der später beim Wecken gesucht wird.

```
PROCEDURE P(z: sname);
   (* Die Prozedur arbeitet bei gesperrten *)
   (* Interrupts als unteilbare Operation   *)
VAR i, k: integer;

BEGIN
i:=seek(z); (* Suche Semaphor *)
IF i> maxs THEN "Systemfehler"
ELSE BEGIN
  s[i].sg:=s[i].sg-1;
  IF s[i].sg<0 THEN BEGIN (* schlafen gehen *)
    WITH proz[ego] DO BEGIN
```



```
      zust:=zust + [wartend];
      sema:=z      (* hierauf warten *)
    END (* with *);
    (* suche rechenwilligen Proze"s *)
    (* und setze ihn rechnend       *)
    k:=maxprio+1; (* alle Priorit"aten durchlaufen *)

    REPEAT
      k:=k-1;
      WHILE prio[k]= 0 DO k:=k-1; (* nicht belegt *)
      i:=prio[k]  (* Proze"snummer *)
    UNTIL bereit IN proz[i].zust;
    (* rechenwilliger Proze"s gefunden *)

    WITH proz[i] DO BEGIN (* notieren *)
      zust:=zust - [bereit] + [rechnend];
      ego:=pnr     (* ihm geh"ort die CPU *)
    END (* with *)

  END (* if sg<0 *)
END (* else *)
END (* P *);
```

Es wird also allein in der Prozeßbeschreibung festgehalten, daß der Prozeß wartet und auf welches Semaphor.

```
PROCEDURE V (z: sname);
  (* Die Prozedur arbeitet bei gesperrten *)
  (* Interrupts als unteilbare Operation  *)
VAR i, k: integer;

BEGIN
  i:=seek(z);  (* Suche Semaphor *)
  IF i> maxs THEN  "Systemfehler"
  ELSE BEGIN
    s[i].sg:=s[i].sg+1;
    IF s[i].sg <= 0 THEN BEGIN  (* es wartet jemand *)
      k:=maxprio+1;
      REPEAT
        k:=k-1;
        WHILE prio[k]= 0 DO k:=k-1; (* nicht belegt *)
        i:=prio[k]
      UNTIL (wartend IN proz[i].zust) AND
      (proz[i].sema= z);
      (* Proze"s gefunden *)
      (* der gefundene Proze"s soll weitermachen *)
      proz[i].zust:=proz[i].zust - [wartend] + [rechnend]
      proz[ego].zust:=proz[ego].zust - [rechnend] + [bere
      (* sich selbst die CPU entziehen *)
```

```
        ego:=i  (* der andere darf *)
        (* hier bekommt der geweckte Proze"s die CPU *)
      END (* if sg<= 0 *)
    END (* else *)
  END (*  V  *);
```

Bei dieser Implementierung wird also ein als zu weckender Prozeß sofort rechnend gesetzt auf Kosten desjenigen Prozesses, der die V-Operation ausgeführt hat. Ihm wird die CPU entzogen bis er sie wieder durch einen übergeordneten Verplanungsvorgang bei nächster Gelegenheit erhält. Man kann sich auch eine andere Implementierung der V-Operation vorstellen. Dabei wird der geweckte Prozeß nur „bereit“ gesetzt und der weckende rechnet weiter. Die entsprechenden Zeilen im Programm werden dann durch

```
proz[i].zust:=proz[i].zust - [wartend] + [bereit];
```

ersetzt.

Eine ganz andere Implementierungstechnik trägt nicht in die Prozeßbeschreibung ein, auf welches Semaphor ein Prozeß wartet, sondern legt für jedes Semaphor eine Schlange an, in die alle darauf wartenden Prozesse eingereiht werden. Dabei muß eine dynamische Speicherverwaltung zum Anlegen der Schlange programmiert werden, denn es können ja mehrere Prozesse und zu verschiedenen Zeiten unterschiedlich viele gleichzeitig auf ein Semaphor warten. Andererseits kann ein Prozeß immer nur auf ein einziges Semaphor warten, weshalb die hier vorgestellte Technik gewählt wurde, bei der keine Speicherprobleme auftreten.

Literaturverzeichnis

[AA78] S. Alagic and M.A. Arbib. *The design of well-structured and correct programs*. Springer-Verlag, 1978.

[Age77] T. Agerwala. Some extended semaphore primitives. *Acta Informatica*, pages 201 – 207, 1977.

[BH72] P. Brinch Hansen. A comparison of two synchronizing concepts. *Acta Informatica*, (1):190 – 199, 1972.

[BH73a] P. Brinch Hansen. *Operating System Principles*. Englewood Cliffs, 1973.

[BH73b] P. Brinch Hansen. A reply to comments on ‘a comparison of two synchronizing concepts’. *Acta Informatica*, (2):189 – 190, 1973.

[Bla89] Uyless D. Black. *Data Networks, Concepts, Theory and Practice*. Prentice Hall International, 1989.

[Buz76] J.P. Buzen. Fundamental operational laws of computer system performance. *Acta Informatica*, (7):167 – 182, 1976.

[CA74] R.H. Campbell and Haberman A.N. The specification of process synchronization by path expressions. *Lecture Notes in Computer Science*, (16):89 – 102, 1974.

[CES71] E.G. Coffman, M.J. Elphick, and A. Shoshani. System deadlocks. *Comp. Surv.*, (2):67 – 78, 1971.

[CHP71] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with ‘readers’ and ‘writers’. *CACM*, (14):667 – 668, 1971.

[dB67] N.G. de Bruijn. Additional comments on a problem in concurrent programming control. *CACM*, (10):137 – 138, 1967.

[DB78] P.J. Denning and J.P. Buzen. The operational analysis of queueing network models. *Comp. Surv.*, (10):225 – 261, 1978.

[Den68] P.J. Denning. The working set model for program behavior. *CACM*, 1968.

[Den75] P.J. Denning. *Software Systems Principles*, chapter ‘Virtual Memory’, pages 204 – 257. SRA-Verlag, 1975.

[Den79] P.J. Denning. *Operating Systems, Theory and Practice*, chapter ‘Working sets: then and now’, pages 115 – 148. Lanciaux, D., North Holland, Amsterdam, 1979.

[Dij65] E.W. Dijkstra. Solution of a problem in concurrent programming control. *CACM*, (8):569, 1965.

[Dij68] E.W. Dijkstra. *Programming Languages*, chapter 'Cooperating sequential processes', pages 43 – 112. Academic Press, 1968.

[Dij71] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, (1):115 – 138, 1971.

[Eil69] S. Eilon. A simpler proof of $L = \lambda \cdot W$. *Operations Research*, (17):915 – 916, 1969.

[EMG72] M.A. Eisenberg and M.R. Mc Guire. Further comments on dijkstra's concurrent programming control problem. *CACM*, (15):999, 1972.

[FH76] L. Flon and A.N. Habermann. Toward the construction of verifyable software systems. *SIGPLAN Notices*, (11):141 – 148, März 1976.

[For78] W.S. Ford. Implemantation of a generalized critical region construct. *IEEE Transaction on Software Engineering*, pages 449 – 455, 1978.

[Fre75] P. Freeman. *Software Systems Principles*. SRA-Verlag, 1975.

[Hab69] A.N. Habermann. Prevention of system deadlocks. *CACM*, (12):373 – 377, 385, 1969.

[Hab72] A.N. Habermann. Synchronization of communicating processes. *CACM*, (15):171 – 176, 1972.

[Hab77] A.N. Habermann. *Perspectives on Computer Science*, chapter 'On the Concurrency of Parallel Processes', pages 77 – 90. 1977.

[Hab78] A.N. Habermann. *'Current Trends in Programming Methodology*, chapter 'System Deadlocks', pages 256 – 279. Prentice Hall, 1978.

[HGLS78] R.C. Holt, G.S. Graham, E.D. Lazowska, and M.A. Scott. *Structured Concurrent Programming with Operating Systems Applications*. Addison Wesley, 1978.

[HMK72] C.A.R. Hoare and R.M. Mc. Keag. *Operating Systems Techniques*, chapter 'A Survey of Store Management Techniques', pages 117 – 131, 132 – 151. Academic Press, 1972.

[Hoa72] C.A.R. Hoare. *Operating Systems Techniques*, chapter 'Towards a Theory of Parallel Programming', pages 61 – 71. Academic Press, 1972.

[Hoa73] C.A.R. Hoare. A structured paging system. *The Computer Journal*, (16):209 – 215, 1973.

[Hoa74] C.A.R. Hoare. Monitors, an operating system structuring concept. *CACM*, (17):549 – 557, 1974.

[Hoa75] C.A.R. Hoare. Parallel programming: An axiomatic approach. *Computer Languages*, (1):151 – 160, 1975.

[HofXX] M. Hofri. Prof of a mutual exclusion algorithm — a 'class'ic example. *Operating Systems Review*, (24):18 – 22, 19XX.

[Hol72] R.C. Holt. Some deadlock properties of computer systems. *Comp. Surv.*, (4):179 – 196, 1972.

[HOR89] Joseph L. Hammond and Peter J.P. O' Reilly. *Performance Analysis of Local Computer Networks*. Addison-Wesley Publishing Company, 1989.

[How76] J.H. Howard. Proving monitors. *CACM*, (19):273 – 279, 1976.

[JS80] A.K. Jones and P. Schwarz. Experience using multiprocessor systems — a staus report. *ACM Comp. Surv.*, (12):121 – 165, Juni 1980.

[Kle75] Leonard Kleinrock. *Queueing Systems, Vol I: Theory*. John Wiley & Sons, 1975.

[Kle76] Leonard Kleinrock. *Queueing Systems, Vol II: Computer Applications*. John Wiley & Sons, 1976.

[Knu66] D.E. Knuth. Additional comments on a problem in concurrent programming control. *CACM*, (9):321 – 322, 1966.

[KNU75] H. Krayl, E.J. Neuhold, and C. Unger. *Grundlagen der Betriebssysteme*. de Gruyter, 1975.

[Kob78a] H. Kobayashi. *Current Trends in Programming Methodology*, chapter 'System Design and Performance Analysis using Analytic Models', pages 72 – 114. Prentice Hall, 1978.

[Kob78b] H. Kobayashi. *Modeling and Analysis*. Addison-Wesley, 1978.

[Lag78] K. Lagally. Synchronisation in a layered system. *Lecture Notes in Computer Science*, 60 (Operating Systems):252 – 281, 1978.

[Lam76] L. Lamport. The synchronisation of independent processes. *Acta Informatica*, (7):15 – 34, 1976.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1 – 11, Februar 1987.

[Lit61] John D.C. Little. A proof for the queueing formula: $L = \lambda \cdot W$. *Operations Research*, (9):383 – 387, 1961.

[MB76] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. *CACM*, (19):395 – 404, 1976.

[MD74] S.E. Madnick and J.J. Donovan. *Operating Systems*. Mc Graw-Hill, Tokio, 1974.

[MG65] W.C. Mc Gee. On dynamic program relocation. *IBM Systems Journal*, (4):184 – 199, 1965.

[New79] G. Newton. Deadlock prevention, detection and resolution: An annotated bibliography. *Operating Systems Review*, 13(2):33 – 44, 1979.

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, (6):319 – 340, 1976.

[OG76b] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, (19):279 – 285, 1976.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Inf. Proc. Lett.*, 12(3):115 – 116, Juni 1981.

[Pie76] F. Pieper. Was ist deadlock? *Elektronische Rechenanlagen*, (18):287 – 292, 1976.

[Ric77] L. Richter. *Betriebssysteme*. Teubner Studienbücher, Informatik, 1977.

[Rob90] Thomas G. Robertazzi. *Computer Networks and Systems: Queueing Theory and Performance Evaluation*. Springer Verlag, 1990.

[Sei91] R. Seifert. Ethernet: Ten years after. *Byte*, (16):315 – 319, 1991.

[Sta91] William Stallings. A practical guide to queueing analysis. *Byte*, pages 309 – 316, Feb. 1991.

[Tan89] Andrew S. Tanenbaum. *Computer Networks (2nd ed.)*. Prentice Hall International, 1989.

[Wir66] N. Wirth. A note on 'program structures for parallel processing'. *CACM*, (9):320 – 321, 1966.

[Zim76] H. Zima. *Betriebssysteme*. BI-Verlag, 1976.